INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Steam Boiler Controller Problem in Signal-Coq

Mickaël Kerbœuf, David Nowak, Jean-Pierre Talpin

N°3773

Octobre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3773--FR+ENG

The Steam Boiler Controller Problem in Signal-Coq

Mickaël Kerbœuf, David Nowak, Jean-Pierre Talpin

Thème 1 — Réseaux et systèmes
Projet EP-ATR

Rapport de recherche n° 3773 — Octobre 1999 — 59 pages

Abstract: Among the various formalisms for the design of reactive systems, the SIGNAL-COQ formal approach, i.e. the combined use of the synchronous dataflow language SIGNAL and the proof assistant COQ , seems to be especially suited and practical.

Indeed, the deterministic concurrency implied by the synchronous model on which SIGNAL is founded strongly simplifies the specification and the verification of such systems. Moreover, COQ is not limited to some kind of properties and so, its use enables to disregard what can be checked during the specification stage.

In this article, we underline the various features of this SIGNAL-COQ formal approach with a large scale case study, namely the Steam Boiler problem.

Key-words: reactive systems, synchronous model, verification, model checking, proof assistant, co-induction

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Le problème du steam-boiler en Signal-Coq

Résumé : Parmi les différents formalismes permettant la conception des systèmes réactifs, l'approche formelle SIGNAL-COQ , *i.e.* l'utilisation combinée du langage de programmation synchrone à flot de données SIGNAL et de l'assistant de preuve COQ , semble être particulièrement adaptée.

En effet, la concurrence déterministe induite par le modèle synchrone sur lequel SIGNAL est fondé simplifie fortement la spécification et la vérification de tels systèmes. En outre, COQ n'est pas limité à un certain type de propriétés et ainsi, son utilisation permet de faire abstraction de ce qui peut être vérifié pendant la phase de spécification.

Dans ce document, nous soulignons les différents aspects de cette approche formelle SIGNAL-COQ à l'aide d'une étude de cas issue de l'industrie, à savoir le problème du steam boiler.

Mots-clé : systèmes réactifs, modèle synchrone, vérification, model-checking, assistant de preuve, co-induction

1 Introduction

In this article, we investigate the combined use of the synchronous language SIGNAL and of the proof assistant COQ for specifying and verifying properties of a large scale case study, namely, the steam boiler problem.
This document is divided into two main parts. Firstly, after a short recall of the context, we present the SIGNAL-COQ formal approach for the design of reactive systems. Then, we report the results of specifying and verifying the steam-boiler problem with this method. Notably, we describe our SIGNAL implementation in details.

1.1 Reactive Systems

Reactive systems are often safety critical applications which are in charge of handling multiple interactions with the environment. They are also often composed of several parallel components that cooperate to achieve the expected behaviour. Those components roughly belong to two main process classes. They are either part of the interface with the environment which ensures the exchange of the data, or they carry out processing on these data. Therefore, concurrency is often an important feature that specification and programming formalisms have to take into account.
Another essential aspect, maybe the most important, is reliability. Indeed, reactive systems are often in charge of critical human or economic resources. Thus, they often require formal verification tools when being designed.

Specification Asynchrony is the most natural model for concurrent programming and seems thus to be well adapted for the specification of reactive systems. Asynchronous languages like CSP [14] or ADA [4] are based on this model. Since the programming model of these languages is founded on the interleaving of external events, verification soon becomes a challenging issue.
The synchronous approach to concurrency (which started to be investigated in [16]) enables to avoid this difficulty, disregarding execution and communication durations. It thus enables to specify, verify and simulate a reactive system at a functional level. The *real-time* feature of the system will only be checked at the implementation stage on a particular architecture. Many programming languages like LUSTRE [12], ESTEREL [7, 9] or STATECHARTS [13] are based on this synchronous approach. Namely, SIGNAL [6, 15] is a dataflow language which handles infinite sequences of data: the *signals*.

Verification The most common way to verify safety properties of reactive systems is model-checking, which relies on searching expected properties in a finite model of the system. This method has several advantages. For instance, it is completly automatic and quite fast. Moreover, it produces counterexamples when the verification happens to fail. But the main drawbacks of model checking are the state explosion problem and the restriction of expressible properties to a given, decidable, logic. Indeed, properties that involve parameters

or non linear numerical values cannot be proved directly with a model-checker.
Theorem proving is not limited by those restrictions. On the other hand, this method, which is based on a mathematical logic given by a formal system, requires interaction with a user. The theorem proving process is therefore slower than the model checking one, and it can sometimes be tedious. Theorem provers can be classified according to their degree of automation. Tools like Coq [5] or Lego require a higher interaction with the user and thus they are called *proof assistants*.
Coq is well suited to resolve complex problems. It enables to manipulate formulas of the calculus of inductive constructions [19], extended with co-induction [11], a logical language coming from type theory. Thus, since Coq can handle infinite objects, it is well adapted to represent signals.

1.2 The Steam-Boiler Control Problem

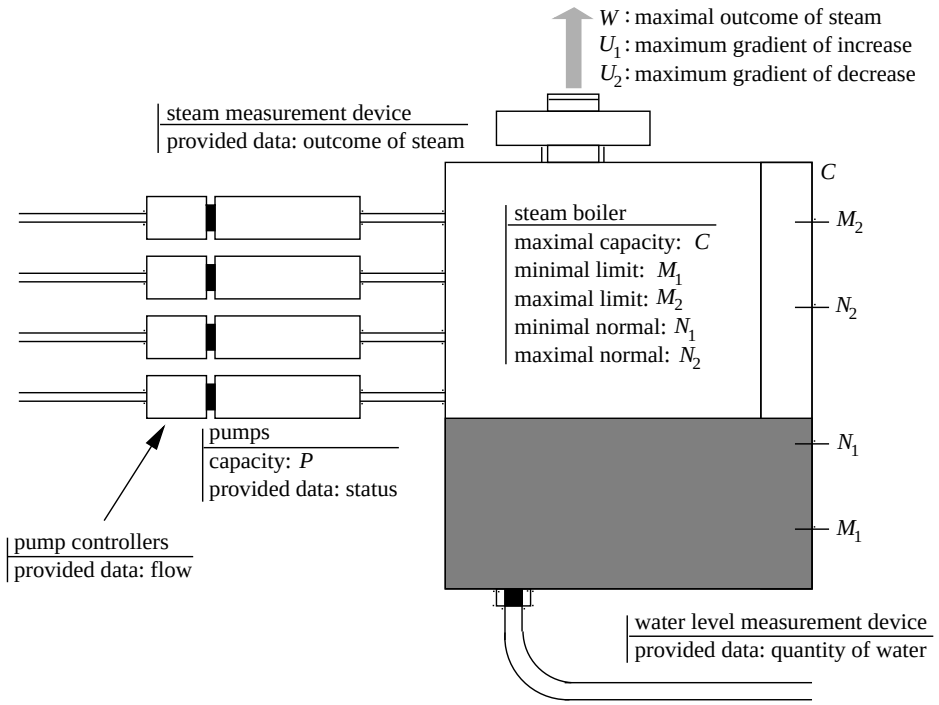


Figure 1: Physical environment

Several formal methods enable to specify and verify a reactive system. In order to compare their strengths and weaknesses, a common case study, i.e. the steam boiler control

specification problem, has been proposed by J.-R. Abrial, E. Börger and H. Langmaack [3, 1, 2]. The original specification is informal, biased to a particular implementation and it contains some ambiguities. Then, some precisions and some decisions about the program and the physical environment must be made at first.

Physical environment The physical environment is composed of several units (FIG. 1). Each one is characterized by physical constants and some of them provide data.

Problem statement The program has to control the level of water in the steam boiler. This quantity has to be neither too low nor too high. Otherwise, the system might be affected. The program also has to manage physical units failures. For that purpose, at every moment, its takes into account the global state of the physical environment which is denoted by an *operation mode*. According to this state, the program decides at each cycle if the system must stop or not, and if not, it activates or deactivates pumps in order to keep the level as much as possible in the middle of the steam boiler.
The specification also gives additional information concerning the physical behaviour of the steam boiler. Namely, new values, called adjusted and calculated values, are proposed. They enable to keep the control of the system, managing a vision of its dynamic, when a measurement device is defective.

We chose this case study because it is well adapted to our aim, i.e. to show the interest of the SIGNAL-COQ formal approach. Indeed, the program has to handle several physical parameters and it may use non linear numerical values. Thus, safety properties cannot be proved directly with a classical model checker.

2 The SIGNAL-COQ Formal Approach

We present in this section the SIGNAL-COQ formal approach for the design of reactive systems which has been proposed in [17]. After an informal recall of the principles of co-induction, we briefly describe the COQ co-inductive axiomatization of SIGNAL , which is also presented in details in [17].

2.1 Co-induction

The common principle of induction enables to underline the characteristics of co-induction. We informaly introduce those ideas in some examples.

Inductive and co-inductive sets A list, i.e. a *finite* sequence of values, is defined in COQ with the `Inductive` command. For example,

```
Inductive natList : Set :=
  Nil : natList
```

```
| Cons : nat->natList->natList.
```

defines a finite list of natural numbers.
Informally, this definition says that a list is either the empty list `Nil`, or `Cons`(x, l) i.e. a value x followed by a list l.
More formally, `natList` defines $\mathcal{L}_{\mathbb{N}}$, the set of the lists of natural numbers. It denotes the smallest fixpoint of a Γ monotonous operator on lists of natural numbers:

$$\Gamma(X) = \{\texttt{Nil}\} \cup \{\texttt{Cons}(x, l) \mid x \in \mathbb{N} \text{ and } l \in X\}$$

We now consider an example of a co-inductive definition. We define a stream, i.e. an *infinite* sequence of values. We use for that purpose the `CoInductive` command in Coq . For example,

```
CoInductive natStream : Set :=
  cons : nat->natStream->natStream.
```

defines an infinite stream of natural numbers.
Informally, this definition says that a stream is always built with the `cons` constructor, i.e. it is always a value followed by a stream.
More formally, `natStream` defines $\mathcal{F}_{\mathbb{N}}$, the set of the streams of natural numbers. It denotes the largest fixpoint of a Δ monotonous operator on streams of natural numbers:

$$\Delta(X) = \{\texttt{cons}(x, F) \mid x \in \mathbb{N} \text{ and } F \in X\}$$

Note that the smallest fixpoint of this operator is the empty set. It is therefore uninteresting in this case.

Recursive and co-recursive functions Coq enables to define recursive and co-recursive functions with the `Fixpoint` operator and the `CoFixpoint` operator. For example,

```
Fixpoint lsum [l:natList] : nat :=
  Cases l of
    Nil => 0
  | (Cons x l) => (plus x (lsum l))
  end.
```

defines a function which calculates the sum of the values of a list.
More formally, this fuction is the smallest fixpoint of the $F_{\texttt{lsum}}$ monotonous function of funtions:

$$\begin{array}{rcl} F_{\texttt{lsum}} : (\mathcal{L}_{\mathbb{N}} \longrightarrow \mathbb{N}) & \longrightarrow & (\mathcal{L}_{\mathbb{N}} \longrightarrow \mathbb{N}) \\ f & \longmapsto & \left\{ \begin{array}{lcl} \texttt{Nil} & \longmapsto & 0 \\ \texttt{Cons}(x, l) & \longmapsto & x + f(l) \end{array} \right. \end{array}$$

Coq also enables to define recursive functions on co-inductive sets. Such functions implements the lazy evaluation principle. Thus, the result of such a function applied to a co-inductive object is computed only during an explicit call of this function. For instance:

```
CoFixpoint map : (nat->nat)->natStream->natStream :=
  [g:nat->nat][F:natStream] Cases F of
    (cons x G) => (cons (g x) (map g G))
  end.
```

applies the g function to each element of a stream. A call to this function on a stream X, infinite by definition, cannot cause its evaluation for every elements of X. In return, g can be applied to the first values of X if these values are explicit. Then, recursive calls must be *guarded*, i.e. the Coq interpreter must be able to find the first value of the stream, to which it will apply the function, in a finite time.
The `map` function defined in this example is the largest fixpoint of the $F_{\texttt{map}}$ monotonous function of funtions:

$$\begin{array}{rcl} F_{\texttt{map}} : ((\mathbb{N} \longmapsto \mathbb{N}) \times \mathcal{F}_{\mathbb{N}} \longmapsto \mathcal{F}_{\mathbb{N}}) & \longrightarrow & ((\mathbb{N} \longmapsto \mathbb{N}) \times \mathcal{F}_{\mathbb{N}} \longmapsto \mathcal{F}_{\mathbb{N}}) \\ f & \longmapsto & ((g, \texttt{cons}(x,l)) \longmapsto \texttt{cons}(g(x), f(g,l))) \end{array}$$

Inductive and co-inductive predicates We can express in Coq the smallest set of elements verifying a relation defined by a set of axioms. For that purpose, we use the `Inductive` operator. For example, the following Mem_n predicate denotes the smallest set of lists of natural numbers that contains a given n:

1. $\forall l \in \mathcal{L}_{\mathbb{N}}, Mem_n(\texttt{Cons}(n,l))$
2. $\forall m \in \mathbb{N}, \forall l \in \mathcal{L}_{\mathbb{N}}, Mem_n(l) \Rightarrow Mem_n(\texttt{Cons}(m,l))$

The Coq specification of this predicate is the following:

```
Inductive Mem [n:nat] : natList->Prop :=
  mem_head: (l:natList)(Mem n (Cons n l))
| mem_tail: (m:nat)(l:natList)(Mem n l)->(Mem n (Cons m l)).
```

On the other side, the `CoInductive` command enables to express the largest set of elements verifying a relation defined by a set of axioms. For example, the following $Eq\_natStream$ predicate denotes the largest set of natural streams that have the same values in the same order:

1. $\forall n \in \mathbb{N}, \forall (X,Y) \in (\mathcal{F}_{\mathbb{N}})^2, Eq\_natStream(X,Y)$
   $\Rightarrow Eq\_natStream(\texttt{Cons}(n,X), \texttt{Cons}(n,Y))$

The Coq specification of this predicate is the following:

```
CoInductive Eq_natStream : natStream->natStream->Prop :=
eqnatstream: (n:nat)(X,Y:natStream)
    (Eq_natStream X Y)->(Eq_natStream (cons n X) (cons n Y)).
```

More generally, co-inductive predicates enable to express properties of *invariance* over streams, whereas inductive predicates enable to express properties of *liveness*.

2.2 Co-inductive axiomatization of SIGNAL in COQ

Proofs of program properties in COQ are built on a set of axioms that denotes the semantic of the language with which the program is written. Thus, in order to have efficient proofs, this semantic has to be well suited to COQ . We use for SIGNAL a variant of the semantic of traces [18], completely described in [17], in which a signal is a co-inductive object.
Informally, in the semantic of traces, signals are represented by a numerical sequence of values and $\perp$ symbols denoting the absence of value. Then, a SIGNAL process is defined as a set of such sequences in respect of some constraints. In the co-inductive variant of this semantic, a signal is a co-inductive set of values and of $\perp$ symbols denoting the absence of value. The COQ specification of such objects is the following:

```
Inductive SValue [U:Set] : Set :=
  absent  : (SValue U)
| present : U->(SValue U).

Definition Signal := [U:Set](Stream (SValue U)).
```

The `absent` constructor denotes the absence of value $\perp$. The parameter U denotes the *type* of the signal. When this signal has a value v at one moment, it is denoted by `present`(v). The co-inductive type `(Stream V)` denotes a stream of V values and it is defined in standard COQ libraries.

This variant is equivalent to the original semantic and has an important advantage: it enables to disregard temporal index which is explicit in numerical sequences.
We can now give the set of axioms that co-inductively models in COQ the operators of the SIGNAL kernel language:

- Instantaneous relations :

```
CoInductive Relation3 [U,V,W:Set; P:U->V->W->Prop] :
  (Signal U)->(Signal V)->(Signal W)->Prop :=
  relation3_a: (X:(Signal U))(Y:(Signal V))(Z:(Signal W))
    (Relation3 P X Y Z)->
    (Relation3 P (Cons (absent U) X)
                 (Cons (absent V) Y)
                 (Cons (absent W) Z))
| relation3_p: (X:(Signal U))(Y:(Signal V))(Z:(Signal W))
    (u:U)(v:V)(w:W)
    (P u v w)->(Relation3 P X Y Z)->
    (Relation3 P (Cons (present u) X)
                 (Cons (present v) Y)
                 (Cons (present w) Z)).
```

- Deterministic merge `default` :

```
CoFixpoint default : (U,V:Set)(Signal U)->(Signal V)->(Signal U+V) :=
  [U,V:Set][X:(Signal U)][Y:(Signal V)] Cases X Y of
    (Cons (present u) X') (Cons _  Y') =>
      (Cons (present (inl ? ? u)) (default X' Y'))
  | (Cons absent X') (Cons (present v) Y') =>
      (Cons (present (inr ? ? v)) (default X' Y'))
  | (Cons absent X') (Cons absent Y') =>
      (Cons ( absent U+V) (default X' Y'))
  end.
```

- Down-sampling `when` :

```
CoFixpoint when : (U:Set)(Signal U)->(Signal bool)->(Signal U) :=
  [U:Set][X:(Signal U)][Y:(Signal bool)] Cases X Y of
    (Cons x X') (Cons (present true) Y') => (Cons x (when X' Y'))
  | (Cons _ X') (Cons _ Y') => (Cons (absent U) (when X' Y'))
  end.
```

- Delay operator :

```
CoFixpoint pre : (U:Set)U->(Signal U)->(Signal U) :=
  [U:Set][u:U][X:(Signal U)] Cases X of
    (Cons absent X') => (Cons (absent U) (pre u X'))
  | (Cons (present v) X') => (Cons (present u) (pre v X'))
  end.
```

SIGNAL equations are naturally transcripted in COQ with this formalization of the language.

3 The steam boiler in SIGNAL-COQ

Because of the flexibility with which the original specification of the steam boiler can be interpreted [3], we first need to make some details more precise on the physical behaviour of the steam boiler and the logical behaviour of it implementation in SIGNAL . Then, we present our proposal in details and the properties that can be checked with COQ .

3.1 Precisions on the original specification

Those precisions concern the behaviour of the physical units, the constitution of the exchanged messages and the reaction they imply, the failure detection, and the management of the vision of the dynamic of the system (fig. 1, p 4).

3.1.1 Distinction between pumps failures and pump controllers failures

A pump controller tells the program if its associated pump is pouring water into the boiler or not. Then, it may enable to check the real state of a pump. For example, when the status of a pump is ON and when this pump tells it, the controller has to confirm it, telling the program if this pump really provides water at this moment. But the controller can also be defective. Thus, it is impossible to distinguish, only with those data, the two following cases for instance:

	pump failure	controller failure	status	state provided by the pump	state provided by the controller	real state
case 1	yes	no	OFF	OFF	ON	ON
case 2	no	yes	OFF	OFF	ON	OFF

Moreover, the specification [3] does not state precisely what *kind* of failure a pump or a controller may have. In the specification, a pump or a controller is defective if its state is inconsistent with its status, and a pump is also defective if it provides a false state. Thus, if the pump and its controller provide a false state but consistent with the pump status, the program cannot detect the failure. For example, the following cases are not distinguishable:

	pump failure	controller failure	status	state provided by the pump	state provided by the controller	real state
case 1	yes	yes	OFF	OFF	OFF	ON
case 2	yes	no	OFF	OFF	OFF	OFF

Because of these ambiguities, the specification states more precisely that in order to detect a controller failure, the program has to know *from elsewhere* that its associated pump works correctly. In such conditions, if the controller provides a state which is inconsistent with the state provided by the pump, the controller is obviously defective.

But how could we be sure that a pump works correctly, i.e. how could we know exactly the *real* state of a pump? We cannot suppose that controllers are infallible because this hypothesis is too restricive with regard to the original specification. Then, in order to know the real state of a pump without controllers information, we must be able to detect precisely which pumps provided water at each cycle according to the level changes.

We first consider a more simple system in which only one pump is involved. Then, suppose that the following conditions hold at instant t:

- the steam measurement device works correctly and provides v_1.
- the water level measurement device works correctly and provides q_1.
- the pump (of capacity P) tells that its state is ON.
- the status of the pump is ON

We now define the following variables:

- minimal accessible level for the next instant $t+1$ if the pump is stopped: $q_{2_{min}}$
- maximal accessible level for the next instant $t+1$ if the pump is stopped: $q_{2_{max}}$
- minimal accessible level for the next instant $t+1$ if the pump provides water: $q'_{2_{min}}$
- maximal accessible level for the next instant $t+1$ if the pump provides water: $q'_{2_{min}}$

We have the following relations:

$$\begin{aligned} q_{2_{min}} &= q_1 - v_1\Delta t - \frac{1}{2}U_1\Delta t^2 \\ q_{2_{max}} &= q_1 - v_1\Delta t + \frac{1}{2}U_2\Delta t^2 \\ q'_{2_{min}} &= q_{2_{min}} + P\Delta t \\ q'_{2_{max}} &= q_{2_{max}} + P\Delta t \end{aligned}$$

In order to be sure that the boiler has actually been supplied in water, the following condition must be verified:

$$[q_{2_{min}}, q_{2_{max}}] \cap [q'_{2_{min}}, q'_{2_{max}}] = \emptyset \tag{1}$$

Indeed, values of this intersection represent levels accessible with or without supply of water. Thus, this constraints enable to be sure that the system has been supplied actually in water if and only if:

$$\begin{aligned} q'_{2_{min}} > q_{2_{max}} &\Leftrightarrow P\Delta t > q_{2_{max}} - q_{2_{min}} && (2) \\ &\Leftrightarrow P\Delta t > \frac{1}{2}\Delta t^2(U_1+U_2) && (3) \\ &\Leftrightarrow \boxed{P > \tfrac{1}{2}\Delta t(U_1+U_2)} && (4) \end{aligned}$$

We now generalize this relation, considering a system with n pumps with capacities p_1 to p_n.
We define the following functions:

$$P_{i(i\in\{1,\ldots,n\})} : \left| \begin{array}{ll} \mathbb{B} \rightarrow & \mathbb{N} \\ b \mapsto & \left\{ \begin{array}{ll} p_i & \text{if } b \\ 0 & \text{if not} \end{array} \right. \end{array} \right.$$

For each combination of pumps (activated or deactivated), we define the cumulated throughput with those functions. Just as in the first case, we settle the following constraint in order to be able to assert that the boiler has actually been supplied in water:

$$\boxed{\forall b_1...b_n \in \mathbb{B}, \sum_{i=1}^{n} P_i(b_i) > \frac{1}{2}\Delta t(U_1 + U_2)} \quad (5)$$

Thus, when this constraint holds, we know how to detect the supply of water by one or several pumps. We have moreover to distinguish exactly which pumps provided water at each cycle. For that purpose, each activated/deactivated combination of pumps must have a unique cumulated throughput. Then the system must also hold the following constraint:

$$\boxed{\forall b_1...b_n, b'_1...b'_n \in \mathbb{B}, (\exists i \in 1..n, b_i \neq b'_i) \Rightarrow \sum_{j=1}^{n} P_j(b_j) \neq \sum_{j=1}^{n} P_j(b'_j)} \quad (6)$$

As a conclusion, to know exactly the real state of each pump disregarding any information coming from the controllers, the system must hold constraints 5 and 6. Now:

- Constraint 5 implies that the exhaust of steam at the exit of the steam boiler is too slow with respect to the troughput of the pumps. Indeed, even the pump which has the lowest capacity must be able to provide in one cycle a higher quantity of water than the maximal quantity of steam which can come out of the boiler. For example, if a pump is defective and cannot be stopped, its repairing must be fast because the level ineluctably rises at each cycle to a critical limit.
- Constraint 6 does not permit to consider pumps which have the same capacity. We can generalize the specification authorizing pumps to have *possibly* different capacities. But this constraint is too restricive with regard to the original specification.
- Those constraints are useless when the steam measurement device or the water level measurement device is defective. Indeed, those constraints are based on real measurement.
- This method makes the controllers useless. Indeed, the states they provide are not needful for the processing.

For all these reasons, we chose to settle the real state of a pump, only regarding its status, the state provided by the pump and the state provided by its controller. Then, we can also settle the real throughput of a pump at each cycle, should it be defective or not (*flow* indicator). This is exactly the choice made in [10], a solution of the steam boiler problem in LUSTRE. The table 1 lists the twelve combinations of possible states and status provided, and the decision on the real states and flow corresponding to each situation. This table was proposed in the LUSTRE implementation. We chose this solution because it seems to be the most reasonable and intuitive one. Moreover, this solution is coherent with the expected role of pump controllers.

- When a pump provides a state which is not coherent with its status, it is defective.

Pump status	State provided by the pump	State provided by the controller	Detected flow	detected failure
OFF	OFF	OFF	no	-
OFF	OFF	ON	no	controller
OFF	ON	OFF	no	pump
OFF	ON	ON	yes	pump
ON	OFF	OFF	no	pump
ON	OFF	ON	yes	pump
ON	ON	OFF	yes	controller
ON	ON	ON	yes	-
SWITCHED-ON	OFF	OFF	no	pump
SWITCHED-ON	OFF	ON	no	pump and controller
SWITCHED-ON	ON	OFF	no	-
SWITCHED-ON	ON	ON	no	controller

Table 1: Pump and controller failure detection

- When a pump provides a state which is coherent with its status, it is not defective.
- When a controller provides a state which is coherent with the status, it is not defective.
- When a controller provides a state which is not coherent with the status, but coherent with the state provided by the pump when this pump is defective, the controller is not defective. This case underlines the interest of a controller. Indeed, we decide here to trust the controller when a pump is defective and when the controller confirms the state provided by its associated pump. Thus, we are able to know the real throughput of this pump.
- When a controller provides a state which is not coherent with the status when the pump works correctly, the controller is defective.
- When a controller provides a state which is not coherent with the status and not coherent with the state provided by the pump when this pump is defective (there is only one case), the controller is also defective.

From those detected failures, we are able to settle the real flow regarding the controller. The flow corresponds to the state provided by the controller when it works correctly. In return, it corresponds to the opposite of this state when the controller is defective.

3.1.2 Precisions about messages

According to the specification [3], some messages must be received at each cycle. We decided to reduce this constraint for some of them. Of course, those messages can still be received at each cycle.

Moreover, the meaning of some messages, and the reaction they imply, must be specified more precisely.

Communication with pumps and controllers The original text does not specify how many pumps or controllers the program can simultaneously communicate with. The messages coming from these physical units are presented with a parameter (e.g. OPEN-PUMP(n)). This parameter may be regarded as a unit identifier. In this case, the program is able to receive only one message of a certain type (e.g. OPEN-PUMP) from one pump or one controller per cycle. In the same way, the program can send only one message of a certain type to one unit. In particular, it can activate or deactivate only one pump at the same time, which is not very realistic.

The parameter of these messages can be considered as the unique identifier of a particular combination of pumps (or controllers). For instance, if the value 2 concerns pumps number 1 and 2, OPEN-PUMP(2) enables to activate them simultaneously. Then, a PUMP-STATE(b) message should be sent back. It could provide the global state of each pump by a (ON / OFF) combination identifier of states (parameter b).

This solution implies that all pumps and controllers have to provide their state after a command concerning only one pump. To reduce this constraint, we decided to use several indexed messages (e.g. OPEN-PUMP1,...,OPEN-PUMP4) corresponding to a particular unit, instead of using one parameterized message (e.g. OPEN-PUMP(n)). This solution increases the number of messages, but it has the advantage of making it possible to manage simultaneously and completely independently each pump and each controller. Moreover, the processing of these messages is some simplified.

Message classes We classify messages in three main classes:

Messages for initialization. We find in this category:

- PROGRAM-READY : This message is a signal sent once during the second phase of the initialization mode to each physical unit.
- STEAM-BOILER-WAITING(i) : The original text does not specify exactly which unit sends this message. Then we decided that each physical unit must send it. We could have therefore decided to create individual communication lines for this message like the messages concerning pumps and controllers. But we do not need in this case to receive simultaneously those messages. Then, we just have to parameterize this message with a unit identifier and we suppose that the line which carries this message is bufferized.

- PHYSICAL-UNITS-READY(i) : This message must be received from each physical unit. It is implemented in a same way than the previous message.

Messages for failure management. We find in this category the following messages. Each message is sent once to the concerned unit. Messages coming from physical units are also sent only once. They must appear in this order:

- Messages for failure detection:
  - PUMPi-FAILURE-DETECTION
  - PUMP-CONTROLi-FAILURE-DETECTION
  - LEVEL-FAILURE-DETECTION
  - STEAM-FAILURE-DETECTION
- Messages for failure acknowledgement:
  - PUMPi-FAILURE-ACKNOWLEDGEMENT
  - PUMP-CONTROLi-FAILURE-ACKNOWLEDGEMENT
  - LEVEL-FAILURE-ACKNOWLEDGEMENT
  - STEAM-FAILURE-ACKNOWLEDGEMENT
- Messages for repairing report:
  - PUMPi-REPAIRED
  - PUMP-CONTROLi-REPAIRED
  - LEVEL-REPAIRED
  - STEAM-REPAIRED
- Messages for repairing acknowledgement:
  - PUMPi-REPAIRED-ACKNOWLEDGEMENT
  - PUMP-CONTROLi-REPAIRED-ACKNOWLEDGEMENT
  - LEVEL-REPAIRED-ACKNOWLEDGEMENT
  - STEAM-REPAIRED-ACKNOWLEDGEMENT

Messages for the processing. They belong to two main classes:

- Information messages:
  - MODE(m) : Sent at each cycle to the physical units.
  - PUMP-STATEi : This message can be received at each cycle but it is compulsory only at the moment which follows a command of deactivation and at the two moments following a command of activation (to secure the successive changes to the status SWITCHED-ON and ON). Moreover this message is compulsory at the initial moment.
  - PUMP-CONTROLi-STATE : This message *must* be present, only when the corresponding PUMP-STATEi message is present.

  - LEVEL(q) : This message must imperatively be present at each cycle. Indeed, the interval of accessible values for the next cycle is calculated with the help of a fixed and preset frequency of measurements. Without this constraint, the vision of the dynamics of the system is false and then, decisions of pump activation or deactivation are not founded.
  - STEAM(v) : For the same reason, this message must imperatively be present at each cycle.
- Action messages :
  - VALVE : This message is sent to the boiler and enables to make the state (open or close) of the valve change for the opposite of its current state.
  - OPEN-PUMPi : This message, sent at the instant t, activates the pump at the instant $(t+1)$ (so the PUMP-STATEi message must be received at $(t+1)$ and $(t+2)$).
  - CLOSE-PUMPi : This message, sent at the instant t, deactivates the pump at the instant $(t+1)$.
  - STOP

In addition to these messages, we introduce a new message H. This message is a pure signal, which is sent every five seconds. This signal is the main clock of the program. All involved signals in the program have a clock which is an sub-clock of H. This signal is supposed to be reliable and makes it possible to detect the absence of compulsory messages.

3.1.3 Behaviour of the physical units in case of a failure

When a failure is notified, the behaviour of the physical unit which is concerned must be specified more precisely.

Measurement devices Measurement devices (steam and water level measurement devices) must always provide a value at each cycle *even if* they are defective. This constraint is not inconsistent with the idea of failure. Indeed, a unit is defective when it *provides* an incoherent measurement with the dynamic of the system. Failure detection of these devices relies on provided measurements and not on their presence. When a measurement misses, a *transmission failure* is detected, which implies the stop of the system in any case.

Thus, when the unit tells the program it is repaired, at the same time, this unit provides a measurement. Since the moment when its failure is detected, the program manages an interval of accessible values for this measurement. Then, it is still possible to estimate (though less precisely) the coherence of this measurement. It is thus possible for the program to detect the failure of a measurement device simultaneously when this unit announces its repairing.

Pumps and contollers When a pump failure is detected, we consider that its state does not change until its repairing. Thus, as the program knows the real flow of a pump when this pump becomes defective, we consider that this flow does not change until the repairing. We also consider that the repairing agent knows the last command which has been given to the pump. Thus, this agent does not only repair the pump. It also puts the state of this pump in a state which is consistent with its expected status when its repairing is notified. Example:

instant	Status	State	Flow	Failure	Repairing agent
t_1	OFF	OFF	no	-	No repairing in progress.
$t_1 + 1$	SWITCHED-ON	OFF	no	YES	Failure notified: beginning of the procedure.
$t_1 + 2$	ON	OFF	no	YES	Repairing procedure in progress.
			. . .		
t_2	ON	ON	no	YES	Procedure finished : pump activation.
$t_2 + 1$	ON	ON	yes	-	Repairing correctly finished.
$t_2 + 2$	ON	ON	yes	-	No repairing in progress.

Thus, it is also possible to dectect a pump failure simultaneously when this pump announces its repairing.

We decided not to give commands to a pump which is defective or whose controller is defective. Then, the flow of such a pump does not change while the concerned physical unit is not correctly repaired. We must make this decision when a controller is defective because without its information, the program is not able to settle neither the real state of its associated pump, nor its real flow. As we consider that this flow does not change until the repairing, the processing is still based on reliable data.

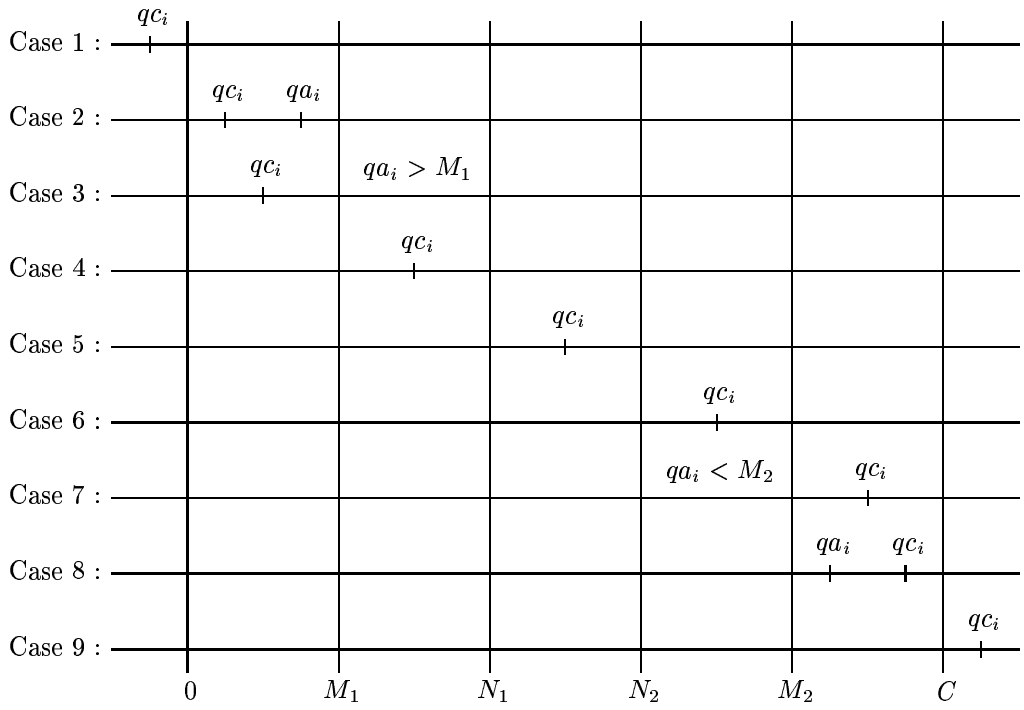
3.1.4 Activation and deactivation of the pumps

In this implementation, the program is only based on the interval of accessible levels for the next cycle, to decide to make the level move up or down, to do nothing if the level is good, or to stop the system because of a critical level.

Decision concerning the level For that purpose, we first consider the following pairs: (qc_1, qa_1) and (qc_2, qa_2). The program computes for each pair the appropriate *decision*. A decision is one of the following values:

- ATT_UP : The level must absolutely move up.

- ATT_DOWN : The level must absolutely move down.
- OK_UP : The level should move up but is not critical yet.
- OK_DOWN : The level should move down but is not critical yet.
- STOP : Critical situation: the system *must* stop.
- NIL : The level is perfect and does not need to be changed.



For each case, the program makes the following decisions:

- Case 1 : STOP. Whatever the current adjusted level may be, the program cannot risk to empty the boiler. Thus, the system must be stopped.
- Case 2 : STOP. The current adjusted level is in a critical zone. The level could still be there at the next cycle because $qc_i < M_1$. As the level cannot risk to remain more than five seconds in this hot zone, the system must be stopped.
- Case 3 : ATT_UP. The current adjusted level is not in a critical zone but it risks to reach such a zone at the next cycle. Then, the level must move up.

- Case 4 : OK_UP. The level is not in a critical zone. Nevertheless, the level is not included in the interval $[N_1, N_2]$. Then, the level might move up.
- Case 5 : NIL. The extremum level for the next cycle is included in the interval $[N_1, N_2]$. Then, the program has to maintain the system in its current state.
- Case 6 : OK_DOWN. The level might move down.
- Case 7 : ATT_DOWN. The current adjusted level is not in a critical zone but it risks to reach such a zone at the next cycle. Then, the level must move down.
- Case 8 : STOP. The current adjusted level is in a critical zone. The level could still be there at the next cycle because $qc_i > M_2$. As in case 2, the level cannot risk to remain more than five seconds in this hot zone. Then, the system must be stopped.
- Case 9 : STOP. Whatever the current adjusted level may be, the level cannot risk to reach the capacity of the boiler. Thus, the system must be stopped.

Then, the program confronts the decisions relating to each extremum in order to make a global decision. The various possible cases are listed in table 2.

To sum up, when a relative decision is STOP, the global decision is STOP. When a relative decision is NIL, the global decision corresponds to the other relative decision. When the kind of a relative decision is ATT, the global decision is STOP when the kind of the other relative decision is also ATT, but in the *opposite* direction, or when the other relative decision is STOP. In the other cases, the global decision corresponds to the direction of this relative decision. Finally, when the kind of the two relative decisions is OK, if they have the same direction, the global decision corresponds to this direction. Otherwise, the program decides not to change the state of the system (decision NIL).

Quantity of water to be provided At this stage, the global decision is final. If this decision is not STOP, the system will not stop in the current cycle because of a critical level. Thus, even if the decision is UP because of a ATT_UP relative decision, and if all the pumps are defective with a zero flow, the system will not stop because it is not in a critical situation. In fact, this situation will probably become critical during the following cycles but the STOP decision will be made only at this moment and not before. Hence, the global decision unifies relative decisions of kind ATT and OK.

Then, when the program decides that the level has to move up or down, it has two main ways to make the level progress towards the expected direction:

- When the global decision is UP, the program can open all the pumps. When the decision is DOWN, it can close all of them. These activation and deactivation commands are only given to pumps which work correctly or whose controllers work correctly.

Decision coming from (qc_1, qa_1)	Decision coming from (qc_2, qa_2)	Global decision
STOP	STOP	STOP
STOP	ATT_UP	STOP
STOP	OK_UP	STOP
STOP	NIL	STOP
STOP	OK_DOWN	STOP
STOP	ATT_DOWN	STOP
ATT_UP	STOP	STOP
ATT_UP	ATT_UP	UP
ATT_UP	OK_UP	UP
ATT_UP	NIL	UP
ATT_UP	OK_DOWN	UP
ATT_UP	ATT_DOWN	STOP
OK_UP	STOP	STOP
OK_UP	OK_UP	UP
OK_UP	NIL	UP
OK_UP	OK_DOWN	NIL
OK_UP	ATT_DOWN	DOWN
NIL	STOP	STOP
NIL	NIL	NIL
NIL	OK_DOWN	DOWN
NIL	ATT_DOWN	DOWN
OK_DOWN	STOP	STOP
OK_DOWN	OK_DOWN	DOWN
OK_DOWN	ATT_DOWN	DOWN
ATT_DOWN	STOP	STOP
ATT_DOWN	ATT_DOWN	DOWN

Table 2: Global decision for the level movement

- According to the global decision and to the current calculated levels, the program calculates the best quantity of water desirable (but not imperious) to be provided for the following cycle.

The first solution is very simple (or even simplistic). However, this radical behaviour implies too strong variations of level. Indeed, if the cumulated throughput of all the pumps enables to fill up the interval $[N_1, N_2]$ in one cycle and if the maximum outcome of steam at the exit of the boiler enables to empty the interval $[N_1, N_2]$ in one cycle, the program risks to activate and deactivate in turn every pumps at each cycle without ever reaching an optimal level.

Then, whe decided to choose the second solution. The program has to calculate the optimal quantity of water that might be provided. For that purpose, we consider the two following cases:

- The global decision is DOWN. Such a decision is implied by the value of qc_2. The program would like to make this value go down to N_2 for the next cycle. For this purpose, it could stop all the pumps. Then, the steam outflow could possibly reduce the level (possibly because some defective pumps could stay activated). Now, considering the minimum outcome of steam, if the reached level remains lower than N_2, the program does not need to deactivate each pump. On the contrary, the pumps should try to provide the difference:

$$\boxed{N_2 - (qc_2 - (vc_1\Delta t - \tfrac{1}{2}U_2\Delta t^2))} \tag{7}$$

  As we want to reduce the level, this quantity is to be reached by *lower* values. Obviously, if this quantity is negative, the program will try not to provide some water. In other words, each non defective pump will be deactivated.

- The global decision is UP. Such a decision is implied by the value of qc_1. The program will try to increase this value toward N_1 for the next cycle. In other words, it will try to provide $(N_1 - qc_1)$ liters of water.
  Moreover, we consider that the quantity of steam at the exit of the boiler will be maximum at the next cycle. This maximum quantity is calculated with the maximum steam outflow and with the maximum gradient of increase of this outflow: $(vc_2\Delta t + \frac{1}{2}U_1\Delta t^2)$. Thus, the program knows the *minimum* quantity of water that it is desirable to provide in order to reach N_1:

$$\boxed{(N_1 - qc_1) + (vc_2\Delta t + \tfrac{1}{2}U_1\Delta t^2)} \tag{8}$$

  As we want to increase the level, this quantity is to be reached by *upper* values. Indeed, in this case, it is better that qc_1 passes beyond N_1 for the next cycle than it remains under N_1.

Thus, with the global decision and the optimal quantity of water to be provided, the program has to calculate the best combination of activated/deactivated pumps in order to reach the aim. Even if this optimal combination is very far from this aim, the system is not stopped because the global decision is (still) not STOP.

3.1.5 State of the system at start

At the initial moment, we decided to create the following constraints:

- All the pumps are in state OFF. This precision is required for the detection of the possible failures of pumps and controllers as soon as the system starts.

- Pumps and controllers must provide their state. The initial status and flows of each pump will be calculated with these data.
- The valve is closed. As VALVE is a pure signal which puts the valve in the opposite state of its current state, its initial state must be specified.

3.1.6 Operation modes

In this implementation, decisions concerned with increasing or decreasing the level of water as well as stopping the system are only made according to the adjusted and calculated values. Such a method enables to avoid the management of the physical state of the units at the level of decision processing. Thus, operation modes *normal, degraded* and *rescue* can be unified because they do not imply a particular processing. For instance, in *normal* mode, all physical units, and namely the water level measurement device, work correctly. In this case, adjusted values qa_1 and qa_2 are the same. More precisely, $qa_1 = qa_2 = q$ with q: value really provided by the level measurement device. In *rescue* mode, this device is defective. Then, adjusted values qa_1 and qa_2 form an interval of possible current levels. In those two cases, pumps activation and deactivation decisions are made in the same way, *independently* of the state of the water level measurement device.

Thus, specified constraints concerning the states of the steam measurement device, of controllers and of pumps in *rescue* mode are useless. In case of failure of these units, the interval formed by adjusted values widens out at each cycle. But as long as this interval does not reach a critical zone, the program does not need to stop the system. Such a method enables to retrieve critical situations as much as possible, without ever reaching a critical state.

Although operation modes *normal, degraded* and *rescue* can be unified, we decided to manage them separately. Indeed, if they do not imply particular actions at the level of decision processing, they can imply particular behaviours of physical units.

3.2 Design and architecture of the SIGNAL implementation

The program is formed by four main processes (FIG. 2). The heart of this architecture is the control process which is in charge of operating the pumps and initially the valve. The three other processes form two kinds of filter which enable to provide to the control process an environment in which it can disregard the management of failures. Notably, this process does not consider possible failures of measurement devices when it decides to activate or deactivate some pumps. Moreover, this process does not consider possible failures of pumps and pump controllers when it decides to stop or not the system.

GESTION_ECHANGES : This process forms the first filter. It is in charge of the detection of transmission failures. For that purpose, it makes sure that the presence of all received messages is not consistent and that all messages whose presence is indispensable at a given cycle are actually received. Then, this process uses our H signal. Messages coming from physical units are passed on only if no transmission failures are detected.

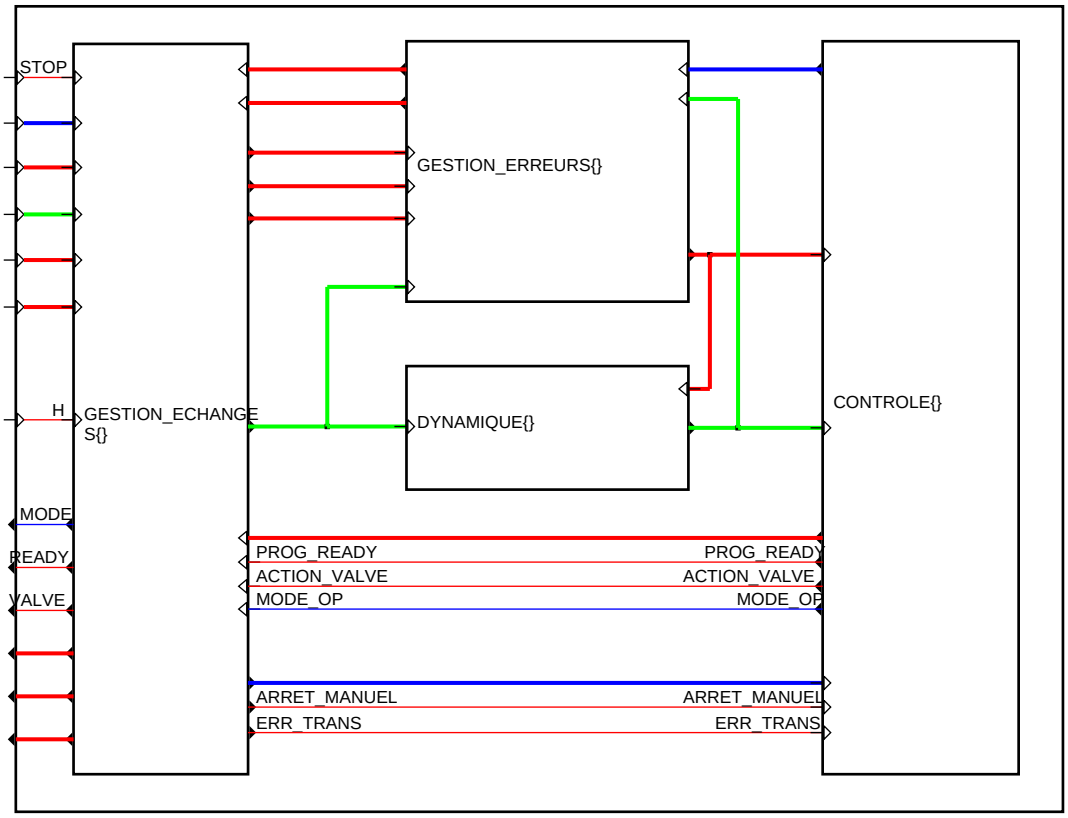


Figure 2: Modular organization of the program

This process is the only user of the H signal. It is also in charge of counting the STOP messages in order to report the manual stop of the system to the control process.

GESTION_ERREURS : This process forms a part of the second filter. It is in charge of the detection of failures of physical units. For that purpose, concerning pumps and controllers, it compares each pump status with each corresponding provided pump and controller states. Concerning measurement devices, it compares the current calculated values with the provided values. When this process detects a failure, it is also in charge of managing the dialogue about failure detection, repairing and corresponding acknowledgements with the concerned unit. Moreover, it provides a global vision of the state of the physical system, reporting at each moment to the control process if

each unit works correctly or not. The real flow of each pump is also defined by this process.

DYNAMIQUE : This process forms the second filter, together with GESTION_ERREURS. It is in charge of providing at each cycle adjusted values of the current measurements and calculated values of the measurements of the following cycle, regarding the state of the measurement devices.
This second filter enables to provide reliable data to the control process on which it will be able to make safe decisions disregarding failure management.

CONTROLE : This process manages operation modes, decides if the system can go on or if it must stop, decides if the level must move up or down in the boiler, calculates the best quantity of water to be provided, and gives commands to the pumps. Those decisions are based on the filtered data coming from the previous processes. Moreover, this process is in charge of the initialization of the system.

The program has a structure of hierarchy. Indeed, SIGNAL enables to create process models which can include sub-models. We thus have an arborescent hierarchy of process models which are included in our implementation (FIG. 3).

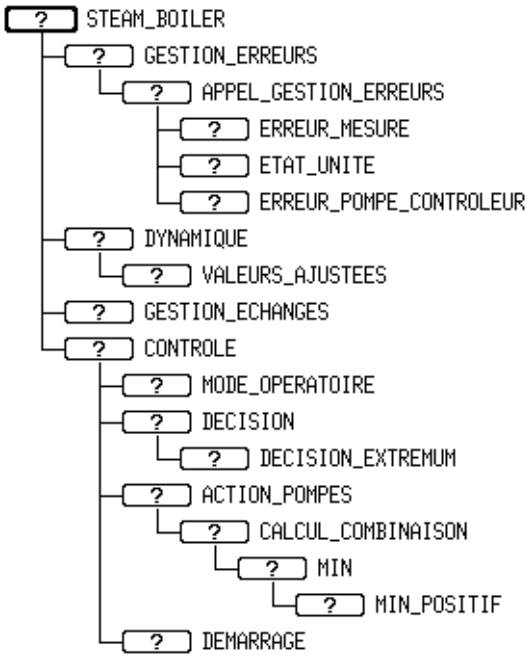


Figure 3: Program's hierarchy

The SIGNAL graphical editor [8] enables to group signals into so called *bundles*[1]. These bundles gather signals of a same kind. Each signal which is specified in the original text has an *internal* version. A message is first received by the GESTION_ECHANGES process. After this filter, those messages are in their internal version. Messages to be sent are also in an internal version. Those messages are actually sent to the physical environment by the GESTION_ECHANGES process. Thus, at the output of the program, those messages are also in their *external* version, i.e. they are as presented in the original specification. In their internal versions, signals are present only if the first process did not detect a transmission failure. Thus, internal signals are always present when they are expected, and only present in this case.
The following signals, gathered in bundles, are used in each level of the program's hierarchy:

- **Initialization messages (received):**
  - STEAM_BOILER_WAITING. Internal version: SB_WAITING
  - PHYSICAL_UNITS_READY. Internal version: UNITS_READY
- **Pumps and controllers state:**
  - PUMP1_STATE,...,PUMP4_STATE. Internal versions: P1_ON,...,P4_ON
  - PUMP_CONTROL1_STATE,...,PUMP_CONTROL4_STATE. Internal versions: PC1_ON,...,PC4_ON
- **Measurement:**
  - LEVEL. Internal version: NIVEAU
  - STEAM. Internal version: VAPEUR
- **Failure detection:**
  - PUMP1_FAILURE_DETECTION,...,PUMP4_FAILURE_DETECTION
    Internal versions: P1_ERR,...,P4_ERR
  - PUMP_CONTROL1_FAILURE_DETECTION,...,
    PUMP_CONTROL4_FAILURE_DETECTION
    Internal versions: PC1_ERR,...,PC4_ERR
  - LEVEL_FAILURE_DETECTION. Internal version: L_ERR
  - STEAM_FAILURE_DETECTION. Internal version: S_ERR
- **Failure acknowledgements:**
  - PUMP1_FAILURE_ACKNOWLEDGEMENT,...,
    PUMP4_FAILURE_ACKNOWLEDGEMENT
    Internal versions: P1_ERR_ACK,...,P4_ERR_ACK

[1] A bundle of signals is not a SIGNAL object. It is just a convention to explain interactions between processes.

  - PUMP_CONTROL1_FAILURE_ACKNOWLEDGEMENT,..., PUMP_CONTROL4_FAILURE_ACKNOWLEDGEMENT
    Internal versions: PC1_ERR_ACK,...,PC4_ERR_ACK
  - LEVEL_FAILURE_ACKNOWLEDGEMENT. Internal version: L_ERR_ACK
  - STEAM_FAILURE_ACKNOWLEDGEMENT. Internal version: S_ERR_ACK

- **Repairing report:**
  - PUMP1_REPAIRED,...,PUMP4_REPAIRED
    Internal versions: P1_REP,...,P4_REP
  - PUMP_CONTROL1_REPAIRED,...,PUMP_CONTROL4_REPAIRED
    Internal versions: PC1_REP,...,PC4_REP
  - LEVEL_REPAIRED. Internal version: L_REP
  - STEAM_REPAIRED. Internal version: S_REP

- **Repairing acknowledgement:**
  - PUMP1_REPAIRED_ACKNOWLEDGEMENT,..., PUMP4_REPAIRED_ACKNOWLEDGEMENT
    Internal versions: P1_REP_ACK,...,P4_REP_ACK
  - PUMP_CONTROL1_REPAIRED_ACKNOWLEDGEMENT,..., PUMP_CONTROL4_REPAIRED_ACKNOWLEDGEMENT
    Internal versions: PC1_REP_ACK,...,PC4_REP_ACK
  - LEVEL_REPAIRED_ACKNOWLEDGEMENT. Internal version: L_REP_ACK
  - STEAM_REPAIRED_ACKNOWLEDGEMENT. Internal version: S_REP_ACK

- **Pump activation and deactivation:**
  - OPEN_PUMP1,...,OPEN_PUMP4. Internal versions: OPEN_P1,...,OPEN_P4
  - CLOSE_PUMP1,...,CLOSE_PUMP4. Internal versions: CLOSE_P1,...,CLOSE_P4

Moreover, the following bundles gather strictly internal messages, i.e. messages which are not intended to be sent to the physical environment:

- **Pump status:** STATUS1,..., STATUS4
- **States of units and flows:**
  - POMPE1_OK,...,POMPE4_OK
  - CONT1_OK,...,CONT4_OK
  - UMDV_OK
  - JAUGE_OK

  - FLOT1,...,FLOT4

- **Calculated and adjusted values:**
  - VAPEUR_CAL_MAX, VAPEUR_CAL_MIN, NIVEAU_CAL_MAX, NIVEAU_CAL_MIN
  - VAPEUR_AJ_MAX, VAPEUR_AJ_MIN, NIVEAU_AJ_MAX, NIVEAU_AJ_MIN

Some messages like PROGRAM_READY, MODE and VALVE are not included in bundles. Their internal version are PROG_READY, MODE_OP and ACTION_VALVE. Messages ARRET_MANUEL and ERR_TRANS are not included in bundles either and are stricly internal. Finally, messages STOP and H do not pass the first filter. Then, they do not have an internal version.

3.2.1 Process of transmissions management

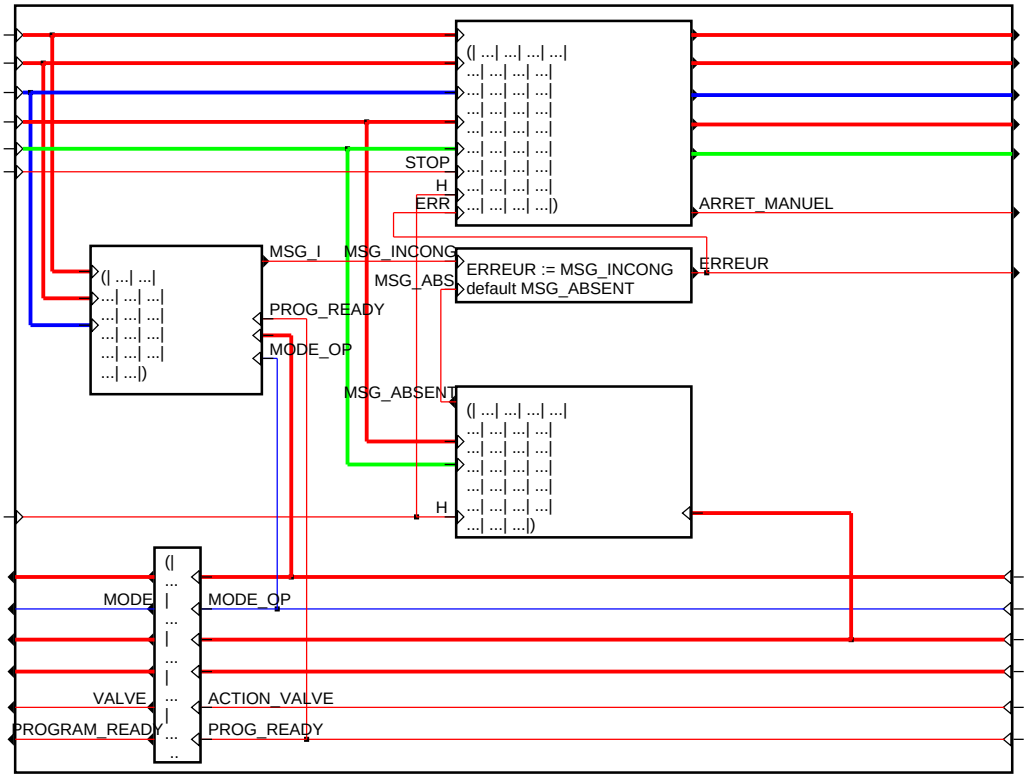


Figure 4: Transmissions management

The GESTION_ECHANGES process consists of five sub-processes (FIG. 4). The first one is in charge of the detection of the absent messages with regard to the H signal. The presence of measurement messages is indispensable at each cycle. Messages that provide pumps and controllers states are compulsory only after an open or a close command. Then, this process manages a counter of expected messages regarding the given commands. Finally, this process makes sure that messages which provide the state of a pump are present when, and only when, messages which provide the state of the corresponding controller are present too.

A second process is in charge of the detection of the incongruous messages. For this purpose, it implements a small automaton which enables to manage the dialogues about failure detection, repairing and corresponding acknowledgements with the concerned units. In each state, it should receive only one kind of message. For example, it has to receive only a failure acknowledgement when a failure was detected at a previous cycle.
This process also manages the dialogue of initialization of the system while being ensured of the good scheduling of the messages constituting this dialogue, and by making sure that they are present only in the initialization phase.

A third process generates a signal which indicates a transmission failure regarding the signals provided by the two preceding processes.

Finaly, two processes are in charge of sending the messages, from inside towards outside without restriction, and from outside towards inside if there are no transmission failure. This last process is also in charge of the detection of the sequences of three consecutive STOP messages with regard to the H signal and of announcing the manual stop of the system.

3.2.2 Process of failures management

The GESTION_ERREURS process (FIG. 5) calls the APPEL_GESTION_ERREURS process (FIG. 6) which calls itself sub-processes corresponding to each physical units. Signals bundles, as they are defined before, gather messages of a same kind. Then, they must be first divided in the GESTION_ERREURS process.

The APPEL_GESTION_ERREURS process uses several instances of three process submodels. The ETAT_UNITE sub model is defined at this level but is used deeper in the hierarchy by the two other models.

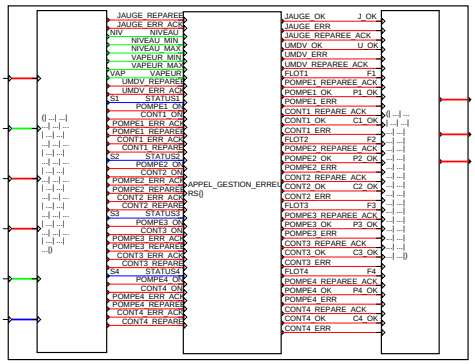


Figure 5: Failures management (1)

The ERREUR_POMPE_CONTROLEUR process model enables to manage dialogues about failure detection, repairing and corresponding acknowledgements with a pump and its controller. It also calculates the real flow of this pump and detects its failures and failures of its controller. Thus, there is one instance of this model per pair of pump and controller.

The ERREUR_MESURE process model enables to manage the dialogue about failure detection, repairing and corresponding acknowledgements with a measurement device. There is one instance of this model for the two measurement devices. These processes use the calculated values for the *current* cycle. Now, calculated values coming from the DYNAMIQUE process correspond to the *next* cycle. So, a process is firstly in charge of delaying theses signals. This process is also in charge of adjusting these delayed calculated values to the physical limits of the system. For instance, when a level value is over the capacity of the boiler, this value is adjusted to the capacity of the boiler. Thus, the program just has to compare only once this value with the measurement provided in order to check that this measurement is consistent with the dynamic of the system and also with the physical constraints of the system.

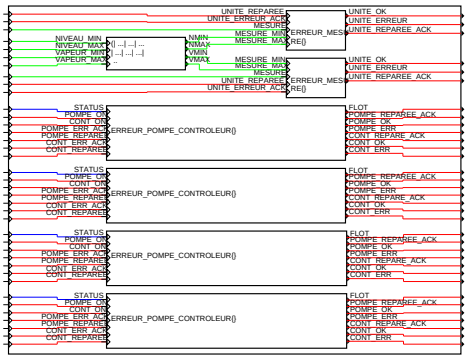


Figure 6: Failures management (2)

State of a physical unit The ETAT_UNITE model is formed of two processes (FIG. 7).

The two models previously defined use the same processing for managing dialogues about failure detection, repairing and corresponding acknowledgements. So, the ETAT_UNITE model enables to factorize this processing.

A first process implements a small automaton which enables to manage the dialogue about failure detection, repairing and corresponding acknowledgements with the concerned unit. This automaton is similar to the one which is implemented in the GESTION_ECHANGES process, but conditions of state transition are different. At this stage, after the first filter, a received message is actually the one which was expected. This process provides the state of the automaton which corresponds to the stage of the dialogue: no failure, failure detected,

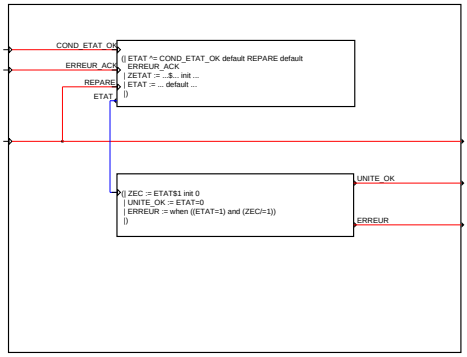


Figure 7: State of a unit

failure detection acknowledged, repairing done.

A second process enables to provide the state (defective or not) of a physical unit independently of the stage of the dialogue with this unit. This process is also in charge of sending the repairing acknowledgements and reporting failures to this unit.

Failures of a measurement device The ERREUR_MESURE process is formed of two processes (FIG. 8).

The first process enables to makes sure that a provided measurement is consistent with the interval of calculated values for the current cycle. Thus, it settles a sufficient condition to detect a failure. But this condition is not necessary because a unit is still defective if its repairing is not finished yet, even if it provides a correct value.

Then, using with this process an instance of the ETAT_UNITE model, the program settles a necessary and sufficient condition to tells the control process at each cycle if a unit works correctly or not.

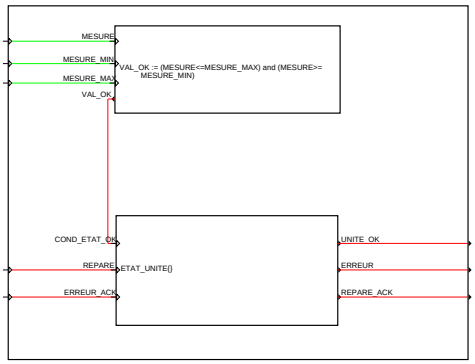


Figure 8: Failures of a measurement device

Failures of pumps an controllers The ERREUR_POMPE_CONTROLEUR process is formed of three processes (FIG. 9).

A first process enables to settle the consistency of the states provided by a pump and its controller. For that purpose, it uses the status of this pump provided by the control process. It directly implements the table 1. Then, like the ERREUR_MESURE process, this process settles sufficient conditions to detect failures of a pump and its controller. Moreover, this process settles the real flow of this pump regarding its failures and failures of its controller. When these two units work correctly, the flow is settled as presented in table 1. In the other cases, the flow keeps its previous value.

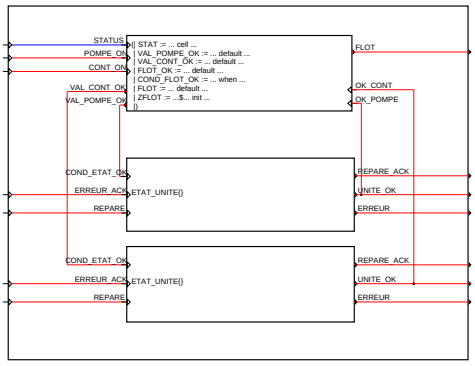


Figure 9: Failures of pumps an controllers

The two other processes are instances of the ETAT_UNITE model. As in ERREUR_MESURE, they enable to manage the dialogue of failure detection and to settle at each moment the state of their associated units regarding the dialogue stage and the conditions of validity provided by the first process.

3.2.3 Dynamic of the system

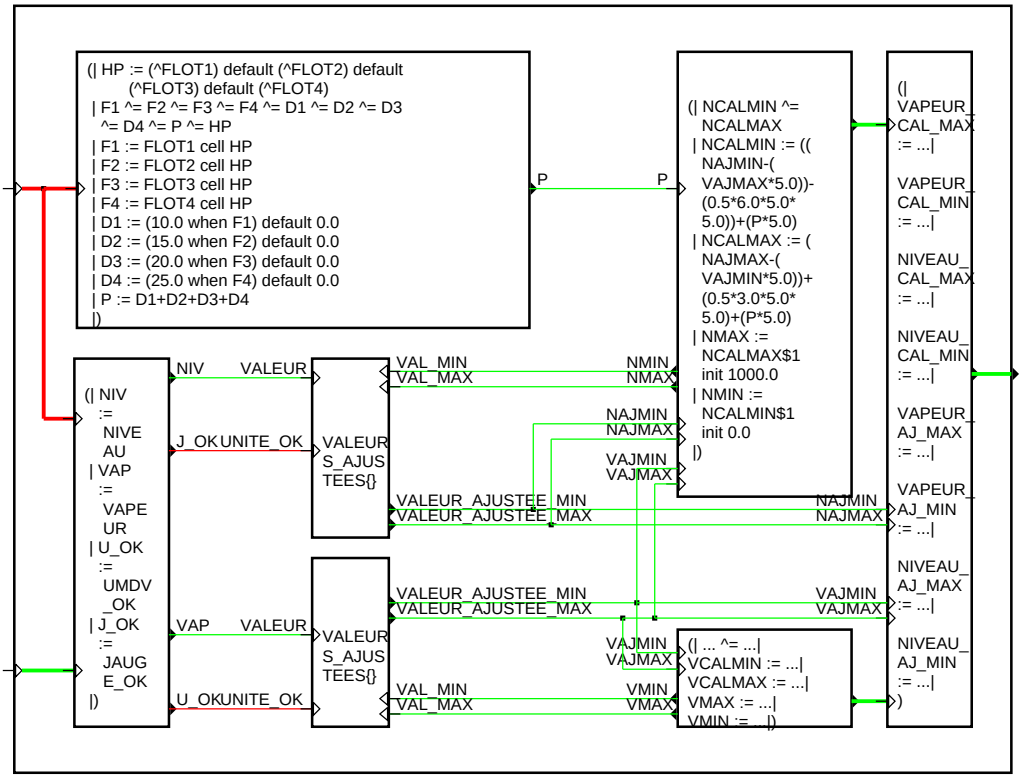


Figure 10: Dynamic

The DYNAMIQUE process is formed of seven sub-processes (FIG. 10). The VALEURS_AJUSTEES sub-model provides the adjusted values of a measurement regarding calculated values concerning the current cycle, current measurements, and the state of the associated unit. In order to use the two instances of this model, a processus is at first in charge of the extraction of signals that carry the states of the physical units and signals that carry the measurements from the bundles.

Two processes provide the calculated values of the outcome of steam and of the water level. For that purpose, they use the current adjusted values and the cumulated throughput of the pumps which is provided by the process placed on the higher left corner. Those processes directly implement the equations given in the original specification. Meanwhile, the term of these equations that concerns the cumulated throughput of the pumps is here simplified because this throughput is settled by the GESTION_ERREURS process.
Those two processes also provide the delayed calculated values that are used to settle adjusted values.

Finally, at the output, a process is in charge of the constitution of the bundle that gathers adjusted and calculated values.

3.2.4 Control process

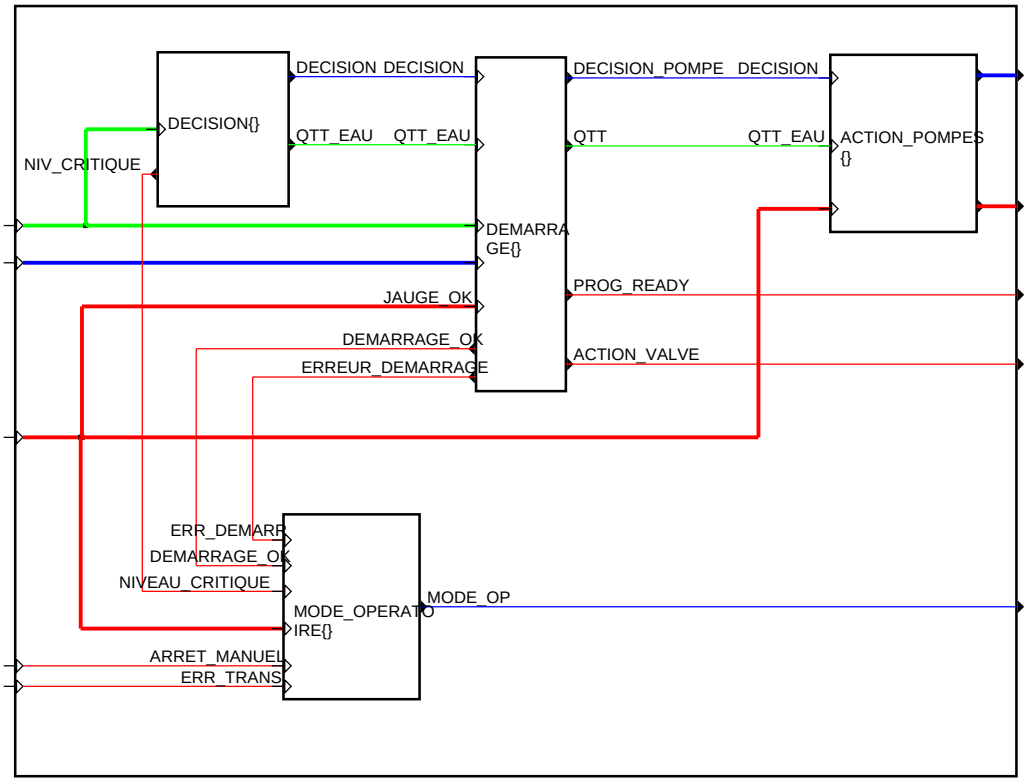


Figure 11: Control process

The DECISION process uses values coming from the DYNAMIQUE process, and only these values, to assess if the system is likely to reach a critical level, in which case it must be stopped. If it is not the case, it decides if the level might move up, down or not move at all. If necessary, it also provides the optimal quantity of water to be provided.

The ACTION_POMPES process settles which pumps must be activated or deactivated, regarding the states of each pump/controller pair, the optimal quantity of water to be provided, and the direction (up or down) of the expected move of the level.

The DEMARRAGE process is in charge of the initialization of the system. Notably, it detects fatal errors during the first step of the initialization mode. It is also in charge of the initial dialogue between the program and the physical units. This process short-circuits the relation between DECISION and ACTION_POMPES. Thus, during the initialization mode, when DECISION asks for a going down of the water level, it operates the valve and it tells ACTION_POMPES to close the pumps. When DECISION asks for a going up of the water level, it actually passes on this decision to ACTION_POMPES. But in this case, the optimal quantity of water is calculated with regard to the middle of the interval $[N_1, N_2]$.
As soon as the level is correct, this process sends a DEMARRAGE_OK message which indicates that the initialization is correctly finished. Then the operation mode changes and the DEMARRAGE process stops to interfere between DECISION and ACTION_POMPES.

Finally, the MODE_OPERATOIRE process provides at each cycle the current operation mode with regard to the states of the physical units and the critical messages that make the program enter the emergency stop mode.

Decision process The DECISION process is formed of five sub-processes (FIG. 12).

The DECISION_EXTREMUM process submodel directly implements the principle of relative decision to an extremum of level suggested in section 3.1.4 (p. 17). The process which is placed on the left of the two instances of this model enables to extract the pairs (qc_1, qa_1) and (qc_2, qa_2) from the signals bundle that contains the adjusted and calculated values coming from the DYNAMIQUE process.

Using these relative decisions, a process calculates the global decision according to the table 2 (p. 20).

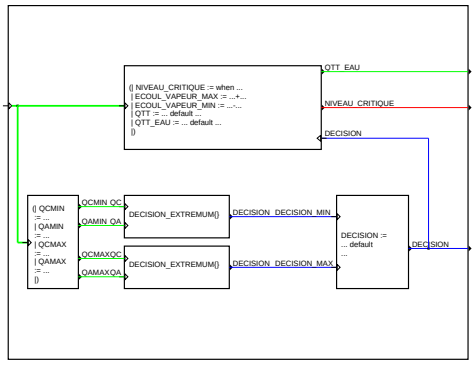


Figure 12: Decision

Finally, a process provides a critical signal when the global decision is STOP and also provides the optimal quantity of water according to the equations 7 (p. 21) and 8 (p. 21).

Pumps operation The ACTION_POMPES process uses one instance of a process sub-model and two sub-processes (FIG. 13).

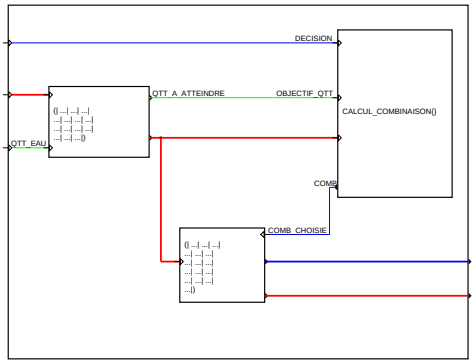


Figure 13: Pumps operation

A first process enables to make received data uniform. Indeed, the pumps and their controllers do not have to provide their states necessarily at each cycle. Now, the program need to know *simultaneously* the state of each pump, of each controller and the real flow of each pump in order to decide which pump will be activated or deactivated at the next cycle. More precisely, the process has to make such decisions when the DECISION process provides a value concerning the quantity of water. For that purpose, using the `cell` command, it creates signals which are syncronous with the QTT_EAU signal. These signals carry the real flow of each pump and the state of each *pair* of pumps and controllers (according to the principle exposed in the section 3.1.3, p. 17, only the pumps which work correctly and whose controller is not defective are *operationable*). Since the process knows the operationability of each pump, it can settle the minimum cumulated throughput for the next cycle. Then, it adjusts the quantity of water to be provided which was sent by the DECISION process. The program has now to reach this new quantity only with the operationable pumps.

The CALCUL_COMBINAISON model calculates the optimal combination of activated/deactivated pumps regarding the global decision and the information coming from the preceding process.

Finally, a third process sends the commands of activation or deactivation of pumps, regarding the combination provided by the instance of the preceding model. It takes into account the operationability of the pumps[2] and the current state of each pump. Thus a pump which is not operationable does not receive commands. An operationable pump receive a command only when it is in the opposite state of its expected state in the combination. As this process gives commands to the pumps, it also manages their status.

[2]The provided combination itself is calculated taking into account the operationability of the pumps. For example, if the first pump is not operationable and open, the combination is necessary a combination whose first pump is activated.

Computation of the optimal combination The CALCUL_COMBINAISON process uses one instance of a process sub-model and two sub-processes (FIG. 14).

A first process calculates the quantity of water that could be provided with each possible combination of pumps. For that purpose, it manages a set of signals which correspond to each combination. At each cycle, these signals are present only if the current combination of pumps states is able to become the corresponding combination at the next cycle. The name of these signals syntactically denotes the associated combination. Thus, O represents an open pump and F a closed one. For instance, the O1O2O3F4 signal is associated with the combination in which all pumps are open except the last one.

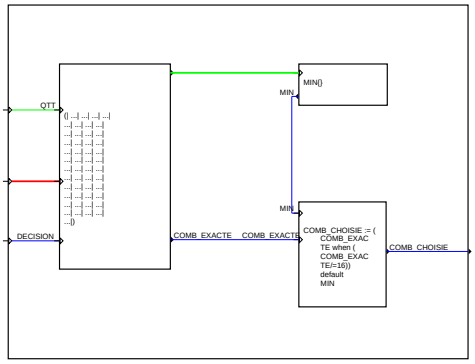


Figure 14: Computation of the combination

In order to illustrate the use of these signals, we now consider the following example: O1F2O3F4 has a value which is: $(P_1 + P_3)\Delta t - qtt$ with *qtt*: quantity of water to be provided. This value is the difference of quantity of water between what the pumps could provide in such a configuration and what the program wants to provide. But this signal is present, with this value, only when the following conditions hold:

- The pump 1 is operationable or it has not a zero flow.
- The pump 2 is operationable or it has a zero flow.
- The pump 3 is operationable or it has not a zero flow.
- The pump 4 is operationable or it has a zero flow.

Thus, signals which are associated with combinations in which some non-operationable pumps are in the opposite state of their current state are not defined. Thereby, these combinations are forbiden.
Remark: Among the sixteen possible combinations, at least one of them is not forbiden. Indeed, in the worst case, all pumps are not operationable but the signal that corresponds to the current configuration is actually defined.
Among these sixteen values, the program looks for a possible zero value. Such a value means that the corresponding combination provides exactly the expected quantity of water. The COMB_EXACTE signal carries the number of this combination when it exists. In the other cases, this signal provides the value NO.

At this stage, the process looks for the best combination but it knows that it cannot provide exactly what is expected. Then, the process compares the different values of the

combinations signals. Now these signals are not synchronous. So it over-samples them with the value 0 which is not significant yet because no exact combinations were found. Thus, sixteen new synchronous signals are created. They either have the value 0 or they have the value of their associated signals of kind x1x2x3x4 when they are defined and when the decision is UP (in this case the program looks for the smallest strictly positive value because the expected quantity of water is to be reached by *upper* values). In the other cases, i.e. when their associated signals of kind x1x2x3x4 are defined and when the decision is DOWN, they have the opposite values to their associated signals. Indeed, in this case, the program looks for the greatest strictly negative value because the expected quantity of water is to be reached by *lower* values. Thus, providing the opposite of the combinations values, the processing is the same as in the first case. This signals are then provided to the MIN process which calculates actually the optimal combination.

The MIN process is a monochronous process. It finds the smallest strictly positive value among its sixteen input signals. If no input value is positive, it finds the greatest negative value. Then it provides the number of the combination which corresponds to the chosen value. If all values are zero, it provides the value NO.

Finally, a process is in charge of providing the chosen combination. This combination is the exact one if such a combination is detected. In the other cases, the chosen combination is the one which is provided by the preceding process. Thus, the program takes indeed into account the combination provided by MIN only when no exact combination is detected. Then, in the MIN process, the value 0 which is used for over-sampling the input signals is actually not significant.

Computation of the minimum The MIN process is formed of five sub-processes (FIG. 15).

This process uses two instances of a sub-model which enables to calculate the strictly positive minimum value among sixteen values. It provides 0 if no input value is strictly positive.

The MIN process provides its input values to the first instance of this model. It provides the opposite of its input values to the second instance. Considering the opposite of the provided result in this case, we thus have the strictly negative maximum value or 0.

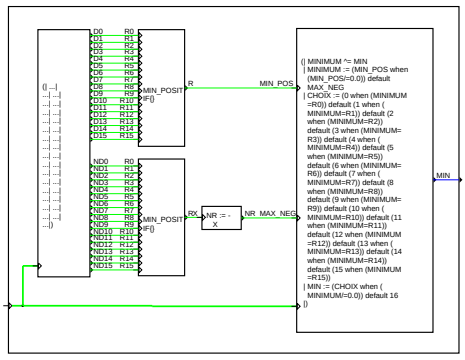


Figure 15: Minimum

Then, a third process settles the chosen value. For that purpose, it first considers the positive minimum value if this value is not 0. Otherwise, it considers the negative maximum value. Then, it settles the number of the corresponding combination if the chosen value is not 0. Otherwise, it provides the NO value.

Start process The behaviour of the start process (DEMARRAGE process) is divided into three stages. The first stage begins at the initialization and ends when all the physical units have sent the STEAM_BOILER_WAITING message.

The second stage begins at this moment and ends when the level in the boiler reaches an expected value. In this stage, the process operates the valve and intercepts the messages coming from DECISION and provides possible changed orders to the ACTION_POMPES process.

The third stage begins when the second one is finished and ends when all the physical units have sent the PHYSICAL_UNITS_READY message. At the first moment of this stage, the process sends the PROGRAM_READY message.

When the third stage is finished, a message is sent in order to report that the initialization of the system is correctly done. Then, the process stops to intercept the messages coming from DECISION and only passes them up to the ACTION_POMPES process. The program operates the valve only during this stage. The valve remains closed after. While the third stage is not finished, a ERREUR_DEMARRAGE message is sent as soon as the failure of a measurement device is detected. This message implies that the program enters the emergency stop mode.

Operation mode management The MODE_OPERATOIRE process provides the current operation mode, regarding critical messages and the states of the physical units. The mode changes from *initialization* to *normal* or *degraded* as soon as this process receives from DEMARRAGE the message which indicates that the initialization of the system is correctly finished.

3.3 Verification

At first, we state in this section a global safety property which is divided into several sub-properties. Then, we present proofs in COQ .

3.3.1 Global safety property

Regarding the specification, a global safety property can be informally stated about the behaviour of the steam boiler:

> "At the moment when a stop condition holds, the system actually stops."

This property can be more precisely stated in the following way:

> "At the moment when a stop condition holds, the program enters the emergency stop mode."

Indeed, according to the specification, in this mode, the physical environment is responsible for taking appropriate actions to stop, and the program also stops.

The first part of this statement deals with the stop conditions of the system. According to our detailed specification, we counted four conditions of this kind:

1. The program consecutively received nb_stop STOP messages from the user.
2. The safety condition about M_1 and M_2 is likely not to be observed anymore.
3. The program detected a transmission failure.
4. The water level measurement device is defective during the initialization of the system.

In our SIGNAL implementation, these conditions are associated with the following critical messages: ARRET_MANUEL, NIVEAU_CRITIQUE, ERR_TRANS and ERREUR_DEMARRAGE. When one of these signals carries a value, the corresponding condition holds and so, the program must stop. This is what we absolutely have to check. Thus, the global safety property can be more formally stated in the following way[3]:

$$\begin{array}{l}\forall t \in \mathbb{N}, \\ \mathsf{ARRET\_MANUEL}(t) = true \vee \mathsf{NIVEAU\_CRITIQUE}(t) = true \\ \vee \mathsf{ERR\_TRANS}(t) = true \vee \mathsf{ERREUR\_DEMARRAGE}(t) = true \\ \Rightarrow \mathsf{MODE\_OP}(t) = stop\end{array} \tag{9}$$

Then, the four following properties must be verified:

$$\forall t \in \mathbb{N}, \mathsf{ARRET\_MANUEL}(t) = true \Rightarrow \mathsf{MODE\_OP}(t) = stop \tag{10}$$

$$\forall t \in \mathbb{N}, \mathsf{NIVEAU\_CRITIQUE}(t) = true \Rightarrow \mathsf{MODE\_OP}(t) = stop \tag{11}$$

$$\forall t \in \mathbb{N}, \mathsf{ERR\_TRANS}(t) = true \Rightarrow \mathsf{MODE\_OP}(t) = stop \tag{12}$$

$$\forall t \in \mathbb{N}, \mathsf{ERREUR\_DEMARRAGE}(t) = true \Rightarrow \mathsf{MODE\_OP}(t) = stop \tag{13}$$

The MODE_OP signal carries the current operation mode at each cycle. It is provided by the MODE_OPERATOIRE sub-process of the CONTROLE process. The operation mode in this process is managed by a little automaton which notably uses the four critical messages. Thus, the four preceding properties can easily be checked.
Now, we have to verify that each critical message is actually present when the condition to which it corresponds holds. For that purpose, we divide properties into two main classes:

[3] In order to state properties that involve signals more legibly, we use the SIGNAL semantic of traces. Thus, a signal of type U is here a function of type $\mathbb{N} \rightarrow (U \cup \{\perp\})$

1. A first class gathers properties that specify the correct behaviour of critical messages, regarding the critical situations to which they correspond.
2. The second class gathers properties that justfy some simplifications or specify the use of some internal signals in the processing.

3.3.2 Coq provable properties

In this section, we only consider properties that cannot be directly verified by a model checker. Indeed, our aim is to show the interest of using a theorem prover like Coq for the verification of reactive systems. Obviously, this approach is uninteresting when all the safety properties can be checked automaticaly and without human interaction. So in the rest of this document, we consider that properties which can be checked by a model checker are actually verified.

Thus, for each process of our Signal implementation, we consider the following properties. They belong to one of the classes defined before and they all involve parameters or non linear numerical values:

1. GESTION_ECHANGES
   When the program receives nb_stop STOP messages from the user, it actually stops. The CPT signal in this process provides the number of consecutive synchronous moments between the signals STOP and H, the main clock. So when this signal provides the value nb_stop, a critical ARRET_MANUEL message must be sent:

   $$\forall t \in \mathbb{N}, \mathsf{CPT}(t) = nb\_stop \Rightarrow \mathsf{ARRET\_MANUEL}(t) = true$$

   Reciprocally, the system manually stops only when the program receives nb_stop STOP messages from the user:

   $$\forall t \in \mathbb{N}, \mathsf{ARRET\_MANUEL}(t) = true \Rightarrow \mathsf{CPT}(t) = nb\_stop$$

   Moreover, the behaviour of the CPT signal is actually the expected behaviour. In order to state this property, we informally define a predicate called C_sc such that $C\_sc(n, X, Y, C)$ is true if, and only if, the numerical signal C is a counter of the consecutive synchronous moments between the signals X and Y with n, its initial value. Thus, we have the following property:

   $$C\_sc(0, \mathsf{STOP}, \mathsf{H}, \mathsf{CPT})$$

   Those properties correspond to the first part of the global safety property.

2. GESTION_ERREURS
   When the water level measurement device is not defective, it provides a value which is consistent with the dynamic of the system, i.e. this value belongs to the interval of calculated levels for the current cycle. The JAUGE_OK signal provides at each cycle the state of this measurement device:

$$\forall t \in \mathbb{N}^*, \mathsf{JAUGE\_OK}(t) = true$$
$$\Rightarrow \mathsf{NIVEAU}(t) \in [\mathsf{NIVEAU\_CAL\_MIN}(t-1), \mathsf{NIVEAU\_CAL\_MAX}(t-1)]$$

When the steam measurement device is not defective, it provides a value which is consistent with the dynamic of the system, i.e. this value belongs to the interval of calculated steam outcomes for the current cycle. The UMDV_OK signal provides at each cycle the state of this measurement device:

$$\forall t \in \mathbb{N}^*, \mathsf{UMDV\_OK}(t) = true$$
$$\Rightarrow \mathsf{VAPEUR}(t) \in [\mathsf{VAPEUR\_CAL\_MIN}(t-1), \mathsf{VAPEUR\_CAL\_MAX}(t-1)]$$

When the water level measurement device is not defective, it provides a value which is consistent with the physical features of the boiler, i.e. this value is positive and is lower than the capacity:

$$\forall t \in \mathbb{N}, \mathsf{JAUGE\_OK}(t) = true \Rightarrow \mathsf{NIVEAU}(t) \in [0, C]$$

When the steam measurement device is not defective, it provides a value which is consistent with the physical features of the boiler, i.e. this value is positive and is lower than the maximal outcome of steam at the exit of the boiler:

$$\forall t \in \mathbb{N}, \mathsf{UMDV\_OK}(t) = true \Rightarrow \mathsf{VAPEUR}(t) \in [0, W]$$

Those properties enable to justify the use of the signals JAUGE_OK and UMDV_OK in the processing, notably in the DECISION process.

3. DYNAMIQUE
   The adjusted values of the steam outcome and the level of water form intervals. When the corresponding measurement devices work correctly, those intervals are reduced to single points. The signals VAPEUR_AJ_MIN, VAPEUR_AJ_MAX, NIVEAU_AJ_MIN and NIVEAU_AJ_MAX correspond to the values va_1, va_2, qa_1 and qa_2 of the specification:

$$\forall t \in \mathbb{N}, \mathsf{VAPEUR\_AJ\_MIN}(t) \leq \mathsf{VAPEUR\_AJ\_MAX}(t)$$

$$\forall t \in \mathbb{N}, \mathsf{NIVEAU\_AJ\_MIN}(t) \leq \mathsf{NIVEAU\_AJ\_MAX}(t)$$

The calculated values of the steam outcome and the level of water form strict intervals. The signals VAPEUR_CAL_MIN, VAPEUR_CAL_MAX, NIVEAU_CAL_MIN and NIVEAU_CAL_MAX correspond to the values vc_1, vc_2, va_1 and va_2 of the specification. These values involve non linear numerical terms:

$$\forall t \in \mathbb{N}, \mathsf{VAPEUR\_CAL\_MIN}(t) < \mathsf{VAPEUR\_CAL\_MAX}(t)$$

$$\forall t \in \mathbb{N}, \mathsf{NIVEAU\_CAL\_MIN}(t) < \mathsf{NIVEAU\_CAL\_MAX}(t)$$

Those properties enable to justify the table 2 (p. 20) which lists the relative decisions. Indeed, this table does not list all the possible combinations of decisions. We thus have to prove that the combinations which are missing in the table are not possible according to the definitions of calculated and adjusted values.
Moreover, those properties enable to simplify proofs that involve calculated and adjusted values.

4. CONTROLE

   - ACTION_POMPES
     The best quantity of water to be reached which is adjusted regarding the states of the pumps and the controllers is at most equal to the quantity of water initially required:

     $$\forall t \in \mathbb{N}, \mathsf{QTT\_A\_ATTEINDRE}(t) \leq \mathsf{QTT\_EAU}(t)$$

   - DECISION
     The properties of this process correspond to the different critical level cases which are listed in the table 2 (p. 20).
     If the calculated levels are over the physical limits of the boiler, the system stops:

     $$\forall t \in \mathbb{N}, \mathsf{NIVEAU\_CAL\_MIN}(t) < 0 \Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = true$$

     $$\forall t \in \mathbb{N}, \mathsf{NIVEAU\_CAL\_MAX}(t) > C \Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = true$$

     If the calculated and adjusted levels are at the same time in a critical zone, the system stops:

     $$\forall t \in \mathbb{N},$$
     $$\mathsf{NIVEAU\_CAL\_MIN}(t) \in [0, M_1] \wedge \mathsf{NIVEAU\_AJ\_MIN}(t) \leq M_1$$
     $$\Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = true$$

     $$\forall t \in \mathbb{N},$$
     $$\mathsf{NIVEAU\_CAL\_MAX}(t) \in [M_2, C] \wedge \mathsf{NIVEAU\_AJ\_MAX}(t) \geq M_2$$
     $$\Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = true$$

     When the calculated levels are at the same time in different critical zones, the system stops:

     $$\forall t \in \mathbb{N},$$
     $$\mathsf{NIVEAU\_CAL\_MIN}(t) \in [0, M_1] \wedge \mathsf{NIVEAU\_CAL\_MAX}(t) \in [M_2, C]$$
     $$\Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = true$$

     If the interval of the calculated levels does not have a common intersection with the critical zones, the program does not stop because of a critical level:

     $$\forall t \in \mathbb{N},$$
     $$\mathsf{NIVEAU\_CAL\_MIN}(t) > M_1 \wedge \mathsf{NIVEAU\_CAL\_MAX}(t) < M_2$$
     $$\Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = \bot$$

     When only the calculated minimum level is in a critical zone, the situation is not critical and then it does not imply the stop of the system:

$\forall t \in \mathbb{N}$,
$\mathsf{NIVEAU\_CAL\_MIN}(t) \in [0, M_1] \wedge \mathsf{NIVEAU\_AJ\_MIN}(t) > M_1$
$\wedge \mathsf{NIVEAU\_CAL\_MAX}(t) < M_2 \Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = \bot$

When only the calculated maximum level is in a critical zone, the situation is not critical and then it does not imply the stop of the system:

$\forall t \in \mathbb{N}$,
$\mathsf{NIVEAU\_CAL\_MAX}(t) \in [M_2, C] \wedge \mathsf{NIVEAU\_AJ\_MAX}(t) < M_2$
$\wedge \mathsf{NIVEAU\_CAL\_MIN}(t) > M_1 \Rightarrow \mathsf{NIVEAU\_CRITIQUE}(t) = \bot$

Those properties correspond to the second part of the global safety property.

- Processus DEMARRAGE
  When the system actually starts, the level is correct (at this moment, the water level measurement device is not defective and then, adjuted levels are equal to the provided measurement):

  $\forall t \in \mathbb{N}, \mathsf{PROG\_READY}(t) = true \Rightarrow \mathsf{NIVEAU\_AJ\_MIN}(t) \in [N_1, N_2]$

3.3.3 Proofs in Coq

Several of the preceding properties have been proved in Coq . As we used the co-inductive axiomatization of Signal in Coq , we could not express these properties using temporal index. Then, we used predicates of temporal logic for that purpose. The Coq specifications of those predicates are in appendix A. The Coq statements of proved properties are in appendix B.

Using the Signal implementation of the steam boiler, some features of proofs in Coq are now presented in the rest of this section.

Generalization of properties for (co-)recursive proofs A property that involves particular values cannot be directly (co-)recursively proved. Indeed, a more general property must be stated at first with non instanciated parameters. Additional hypotheses about these formal parameters can also be stated. In order to illustrate this, we define the following function:

$$f_1 : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \begin{cases} 1 & \text{if } n = 0 \\ n f_1(n-1) & \text{otherwise} \end{cases} \end{array}$$

We now consider the following function:

$$f_2 : \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (n, r) & \mapsto & \begin{cases} r & \text{if } n = 0 \\ f_2(n-1, nr) & \text{otherwise} \end{cases} \end{array}$$

It seems to be obvious that:

$$\forall n \in \mathbb{N}, f_1(n) = f_2(n, 1)$$

But such a proof is made by induction on n. The proof is impossible because of the parameter 1. It is too limitative for the recurrence hypothesis, which thus cannot be applied. Then, we have to prove at first:

$$\forall (n, r) \in \mathbb{N}^2, r f_1(n) = f_2(n, r)$$

Then, the expected property corresponds to the case where r is instanciated with 1.

We have the same problem with the properties that involve the `pre` operator. Indeed, this operator need a value for initialization. So, we open a *section*, which in Coq enables to define local variables and local hypotheses. In this section, we state a general property where the `pre` operators only contain formal parameters. If necessary, we state in this section new hypotheses about these formal parameters. Then, we prove the property in this extended environment. When we close this section, the proved theorems remain. So we apply these theorems to the initial property. Then, we only have to prove that the particular values in `pre` are consistent with the required constraints.

Specification of co-inductive predicates and co-recursive proofs We sometimes have to create co-inductive predicates to state some properties. But the co-recursive proof of such properties can be difficult. For example, we consider the C_sc predicate which was previously informally presented: $C\_sc(n, X, Y, C)$ is true if, and only if, the numerical signal C is a counter of the consecutive synchronous moments between the signals X and Y with n, its initial value. This predicate enables to state a property concerning the correct behaviour of our CPT signal: $C\_sc(0, \mathsf{STOP}, \mathsf{H}, \mathsf{CPT})$.
The following axioms co-inductively define C_sc:

1. $c\_sc\_aa : \forall X \in \mathcal{F}_{(U \cup \{\bot\})}, \forall Y \in \mathcal{F}_{(V \cup \{\bot\})}, \forall C \in \mathcal{F}_{(\mathbb{N} \cup \{\bot\})}, \forall n \in \mathbb{N},$
   $C\_sc(n, X, Y, C) \Rightarrow C\_sc(n, \mathtt{Cons}(\bot, X), \mathtt{Cons}(\bot, Y), \mathtt{Cons}(\bot_{\mathbb{N}}, C))$
2. $c\_sc\_ap : \forall X \in \mathcal{F}_{(U \cup \{\bot\})}, \forall Y \in \mathcal{F}_{(V \cup \{\bot\})}, \forall C \in \mathcal{F}_{(\mathbb{N} \cup \{\bot\})}, \forall n \in \mathbb{N}, \forall v \in V,$
   $C\_sc(n, X, Y, C) \Rightarrow C\_sc(0, \mathtt{Cons}(\bot, X), \mathtt{Cons}(v, Y), \mathtt{Cons}(0, C))$
3. $c\_sc\_pa : \forall X \in \mathcal{F}_{(U \cup \{\bot\})}, \forall Y \in \mathcal{F}_{(V \cup \{\bot\})}, \forall C \in \mathcal{F}_{(\mathbb{N} \cup \{\bot\})}, \forall n \in \mathbb{N}, \forall u \in U,$
   $C\_sc(n, X, Y, C) \Rightarrow C\_sc(0, \mathtt{Cons}(u, X), \mathtt{Cons}(\bot, Y), \mathtt{Cons}(0, C))$
4. $c\_sc\_pp : \forall X \in \mathcal{F}_{(U \cup \{\bot\})}, \forall Y \in \mathcal{F}_{(V \cup \{\bot\})}, \forall C \in \mathcal{F}_{(\mathbb{N} \cup \{\bot\})}, \forall n \in \mathbb{N}, \forall u \in U, \forall v \in V,$
   $C\_sc(n, X, Y, C) \Rightarrow C\_sc(n+1, \mathtt{Cons}(u, X), \mathtt{Cons}(v, Y), \mathtt{Cons}(n+1, C))$

Informally, these four axioms correspond to each possible combination of possible values at each moment for two signals:

- When both signals are absent, the counter has no value and its last value remains the same.
- If one of the two signals is absent when the other is present, the counter's value becomes 0.

- When both signals are present, the counter is incremented.

Then, if $C\_sc(n, X, Y, C)$ is true, the C signal is present when X or Y is present. When X and Y are absent, C is absent too. When C is present, it provides the number of the consecutive synchronous moments between the signals X and Y with n, its initial value. Such a predicate is naturally defined in Coq in the following way:

```
CoInductive C_sc [U,V:Set] :
nat->(Signal U)->(Signal V)->(Signal nat)->Prop :=
 c_sc_aa : (X:(Signal U))(Y:(Signal V))(C:(Signal nat))(n:nat)
           (C_sc n X Y C)
           -> (C_sc n (Cons (absent U) X) (Cons (absent V) Y)
                      (Cons (absent nat) C))
|c_sc_ap : (X:(Signal U))(Y:(Signal V))(C:(Signal nat))(n:nat)(v:V)
           (C_sc n X Y C)
           -> (C_sc 0 (Cons (absent U) X) (Cons (present v) Y)
                      (Cons (present 0) C))
|c_sc_pa : (X:(Signal U))(Y:(Signal V))(C:(Signal nat))(n:nat)(u:U)
           (C_sc n X Y C)
           -> (C_sc 0 (Cons (present u) X) (Cons (absent V) Y)
                      (Cons (present 0) C))
|c_sc_pp : (X:(Signal U))(Y:(Signal V))(C:(Signal nat))(n:nat)(u:U)(v:V)
           (C_sc n X Y C)
           -> (C_sc (S n) (Cons (present u) X) (Cons (present v) Y)
                          (Cons (present (S n)) C)).
```

The use of such a predicate can be difficult in proofs. Intuitively, these axioms enable to infer an expected property at a given moment, regarding an hypothesis about future moments. Then, the use of such a predicate implies a progression from future moments toward past moments. Now, in a co-inductive proof, we suppose that the expected property is verified at t. We prove that axioms can be applied at this moment, and then, we apply the co-recurrence hypothesis in order to prove that the property is also verified in the following moments. In other words, the progression of the proof goes from the present moment toward future moments.

In order to avoid this chaotic iteration in the proof and in order to give a specification of the behaviour of CPT in Coq nearer to its Signal definition, we prefer to use the following function:

$$
\begin{array}{rl}
F: & (\mathbb{N} \times \mathcal{F}_{(U\cup\{\bot\})} \times \mathcal{F}_{(V\cup\{\bot\})} \to \mathcal{F}_{(\mathbb{N}\cup\{\bot\})}) \\
& \longrightarrow (\mathbb{N} \times \mathcal{F}_{(U\cup\{\bot\})} \times \mathcal{F}_{(V\cup\{\bot\})} \to \mathcal{F}_{(\mathbb{N}\cup\{\bot\})}) \\
& f \longmapsto \left\{ \begin{array}{lcl}
(n, \mathtt{Cons}(\bot, X), \mathtt{Cons}(\bot, Y)) & \mapsto & \mathtt{Cons}(\bot, f(n, X, Y)) \\
(n, \mathtt{Cons}(x, X), \mathtt{Cons}(\bot, Y)) & \mapsto & \mathtt{Cons}(0, f(0, X, Y)) \\
(n, \mathtt{Cons}(\bot, X), \mathtt{Cons}(y, Y)) & \mapsto & \mathtt{Cons}(0, f(0, X, Y)) \\
(n, \mathtt{Cons}(x, X), \mathtt{Cons}(y, Y)) & \mapsto & \mathtt{Cons}(n+1, f(n+1, X, Y))
\end{array} \right.
\end{array}
$$

This function enables to define the *cis* function, its largest fixpoint. This new function provides a signal of type $\mathbb{N}$ which corresponds to a counter of the successive synchronous moments of its parameters, with n, its initial value. Proving $CPT = cis(0, STOP, H)$, we prove that CPT has actually the expected behaviour. Such a function is naturally defined in COQ in the following way:

```
CoFixpoint cpt_inst_synch :
(U,V:Set)nat->(Signal U)->(Signal V)->(Signal nat) :=
[U,V:Set][n:nat][X:(Signal U)][Y:(Signal V)]Cases X Y of
 (Cons absent X')      (Cons absent Y')
   => (Cons (absent nat) (cpt_inst_synch n X' Y'))
|(Cons (present _) X') (Cons absent Y')
   => (Cons (present 0) (cpt_inst_synch 0 X' Y'))
|(Cons absent X')      (Cons (present _) Y')
   => (Cons (present 0) (cpt_inst_synch 0 X' Y'))
|(Cons (present _) X') (Cons (present _) Y')
   => (Cons (present (S n)) (cpt_inst_synch (S n) X' Y'))
end.
```

Intuitively, the applying of this function on signals and the co-inductive proof of the expected property have the same progression: from the present moment towards the future moments. Moreover, this COQ specification of CPT gives its functional behaviour and is thus more intuitive.

Syntactical simplifications COQ enables to modify the way it parses and prints objects. Using advanced command, we can thus change the syntax of concrete representations of terms and commands. For instance, the ZArith library provides an axiomatization of $\mathbb{Z}$, and also several lemmas and theorems that enable to solve equations and inequations on $\mathbb{Z}$. We now consider the following statement:

$$(\forall x, y \in \mathbb{Z})(0 \leq x) \Rightarrow (0 < y) \Rightarrow (0 < x + y)$$

Using the ZArith library, the COQ definition of this statement is the following:

```
(x,y:Z)(Zle ZERO x)->(Zlt ZERO y) -> (Zlt ZERO (Zplus x y))
```

Meanwhile, the ZArith library also provides syntactical facilities. Thus, we have an equivalent way to define this statement:

```
(x,y:Z)'0 <= x'->'0 < y'->'0 < x+y'
```

Such a syntax is more intuitive and so, proving equations or inequations on $\mathbb{Z}$ in COQ are much easier.
Thus, it is probably possible to extend the syntax of COQ to the syntax of SIGNAL . We could then naturally define in COQ the SIGNAL equations of a program, without translation.

4 Conclusion

We showed in this article that the SIGNAL-COQ formal approach is well suited for specifying and verifying properties of a reactive system. We applied this design formalism on the steam boiler controller problem. As the original specification of [3] was sometimes informal, we made some description of the physical environment more precise. This problem is well adapted to the evaluation of our formal approach because of its strong safety property which implies the handling of parameters and non linear numerical values.

In spite of the strong implication of the user during the process of proof, it appears that the use of a proof assistant like COQ has many advantages. In addition to the fact that it is not limited to some kinds of properties, it makes it possible to acquire a strong confidence in the program. Moreover, COQ gains with being often used on SIGNAL programs because the encountered problems are often similar. By developing the COQ libraries concerning SIGNAL, we improve the effeciency of the later proofs. Lastly, let us note that it is possible to improve the legibility of properties and proofs by extending the syntax of COQ to the syntax of SIGNAL .

However, this approach is interesting only with properties that cannot be directly proved by a model checker. It is thus advisable to use a proof assistant in complement for these particular properties. So, the unification within a same development framework of model-checking and theorem-proving seems to be a promising prospect.

A Temporal logic in Coq

The Temporal library gathers inductive and co-inductive predicates that define **G**, **F** and their derivative operators.

```
Require Streams.

Implicit Arguments On.

Section Temporal_Section.

Inductive Future1 [U:Set; P:(Stream U)->Prop] : (Stream U)->Prop :=
  here1: (X:(Stream U))(P X) ->(Future1 P X)
| further1: (X:(Stream U))(Future1 P (tl X))->(Future1 P X).

Inductive Future2 [U,V:Set; P:(Stream U)->(Stream V)->Prop] :
  (Stream U)->(Stream V)->Prop :=
  here2: (X:(Stream U))(Y:(Stream V))(P X Y) ->(Future2 P X Y)
| further2: (X:(Stream U))(Y:(Stream V))
    (Future2 P (tl X) (tl Y))->(Future2 P X Y).

CoInductive Globally1 [U:Set; P:(Stream U)->Prop] : (Stream U)->Prop :=
  globally1 : (X:(Stream U))(P X)->(Globally1 P (tl X))->(Globally1 P X).

CoInductive Globally2 [U,V:Set;P:(Stream U)->(Stream V)->Prop] :
(Stream U)->(Stream V)->Prop :=
  globally2 : (X:(Stream U))(Y:(Stream V))(P X Y)
              ->(Globally2 P (tl X) (tl Y))->(Globally2 P X Y).

CoInductive Globally3 [U,V,W:Set;P:(Stream U)->(Stream V)->(Stream W)->Prop] :
(Stream U)->(Stream V)->(Stream W)->Prop :=
  globally3 : (X:(Stream U))(Y:(Stream V))(Z:(Stream W))(P X Y Z)
              ->(Globally3 P (tl X) (tl Y) (tl Z))->(Globally3 P X Y Z).

Lemma G2_andG1 :
(U,V:Set)(P1:(Stream U)->Prop)(P2:(Stream V)->Prop)(X1:(Stream U))(X2:(Stream V))
(Globally2 [X:(Stream U)][Y:(Stream V)](P1 X)/\(P2 Y) X1 X2)
->
(Globally1 P1 X1)/\(Globally1 P2 X2).

Lemma andG1_G2 :
(U,V:Set)(P1:(Stream U)->Prop)(P2:(Stream V)->Prop)(X1:(Stream U))(X2:(Stream V))
(Globally1 P1 X1)/\(Globally1 P2 X2)
->
(Globally2 [X:(Stream U)][Y:(Stream V)](P1 X)/\(P2 Y) X1 X2).

Definition GloballyFuture1 :=
  [U:Set][P:(Stream U)->Prop][X:(Stream U)]
    (Globally1 [Y:(Stream U)](Future1 P Y) X).

Definition FutureGlobally1 :=
  [U:Set][P:(Stream U)->Prop][X:(Stream U)]
    (Future1 [Y:(Stream U)](Globally1 P Y) X).
```

```
Inductive Until1 [U:Set; P,Q:(Stream U)->Prop] : (Stream U)->Prop :=
  q_here1: (X:(Stream U))(Q X)->(Until1 P Q X)
| q_further1: (X:(Stream U))(P X)->(Until1 P Q (tl X))->(Until1 P Q X).

Definition WeakUntil1 : (U:Set)((Stream U)->Prop)->((Stream U)->Prop)->
(Stream U)->Prop:=
  [U:Set][P,Q:(Stream U)->Prop][X:(Stream U)]
    (Until1 P Q X) \/ (Globally1 P X).

End Temporal_Section.
```

B Coq libraries for the steam boiler

Definitions, theorems and proofs are gathered in three libraries. They use the Coq standard libraries and the libraries that provide the co-inductive axiomatization of Signal [17].

B.1 Ext_ZArith

The ZArith library provides several definitions, lemmas and theorems in order to resolve equations on $\mathbb{Z}$. Our Ext_ZArith library is an extent of it. It provides additional lemmas which are used in the solution of equations involved in the properties of the DYNAMIQUE process.

```
Require Classical.
Require Export ZArith.

Implicit Arguments On.

Section Ext_ZArith.

(*--------------------------------------------------------------------------------------*)
(*                                  Conversion bool/Prop                                *)
(*--------------------------------------------------------------------------------------*)

Lemma Zlebool_Zle_true : (n,m:Z)(Zle_bool n m)=true -> (Zle n m).

Lemma Zlebool_Zle_false : (n,m:Z)(Zle_bool n m)=false -> ~(Zle n m).

Lemma Zle_Zlebool_true : (n,m:Z)(Zle n m) -> (Zle_bool n m)=true.

Lemma Zle_Zlebool_false : (n,m:Z)~(Zle n m) -> (Zle_bool n m)=false.

Lemma Zltbool_Zlt_true : (n,m:Z)(Zlt_bool n m)=true -> (Zlt n m).

Lemma Zltbool_Zlt_false : (n,m:Z)(Zlt_bool n m)=false -> ~(Zlt n m).

Lemma Zlt_Zltbool_true : (n,m:Z)(Zlt n m) -> (Zlt_bool n m)=true.

Lemma Zlt_Zltbool_false : (n,m:Z)~(Zlt n m) -> (Zlt_bool n m)=false.
```

```
(*-------------------------------------------------------------------------------------*)
(*                                Lemmas for solutions                                 *)
(*-------------------------------------------------------------------------------------*)

Lemma Zmult_pos_pos : (x,y:Z)(Zlt '0' x)->(Zlt'0' y) -> (Zlt '0' (Zmult x y)).

Lemma Zmult_pos_neg : (x,y:Z)(Zlt '0' x)->(Zgt'0' y) -> (Zgt '0' (Zmult x y)).

Lemma Zplus_pos_pos : (x,y:Z)(Zlt '0' x)->(Zlt'0' y) -> (Zlt '0' (Zplus x y)).

Lemma Zplus_zpos_pos : (x,y:Z)(Zle '0' x)->(Zlt'0' y) -> (Zlt '0' (Zplus x y)).

Lemma Zgt0_Zlt0 : (x:Z)(Zgt x '0')->(Zlt (Zinv x) '0').

Lemma Zge0_Zle0 : (x:Z)(Zge x '0')->(Zle (Zinv x) '0').

Lemma Zlt0_Zgt0 : (x:Z)(Zlt x '0')->(Zgt (Zinv x) '0').

Lemma Zle0_Zge0 : (x:Z)(Zle x '0')->(Zge (Zinv x) '0').

Lemma Zlt_Zle : (x,y:Z)(Zlt x y)->(Zle x y).

Lemma Zgt_Zge : (x,y:Z)(Zgt x y)->(Zge x y).

End Ext_ZArith.
```

B.2 Gestion_ech

The Gestion_ech library gathers definitions and predicates that enable to express properties of the GESTION_ECHANGES process, as also proofs of these properties.

```
Require Arith.
Require EqNat.
Require Temporal.
Require OrderClock.
Require Flot.

Implicit Arguments On.

(*------------------------------ GESTION_ECHANGES PROCESS -------------------------------*)
(*             parameter nb_stop : number of msg. STOP required for the stop.            *)
(*

   | CPT ^= H
   | CPT := ((ZCPT+1) when STOP) default (0 when H)
   | ZCPT := CPT$1 init 0
   | ARRET_MANUEL := when (CPT=nb_stop)

Equations:

 0 | STOP ^< H
```

```
 1 | CPT ^= H
 2 | Cst0 := 0
 3 | Cst0 ^= H
 4 | CPT := ((ZCPT+1) when STOP) default (Cst0 when H)
 5 | ZCPT := CPT$1 init ni    (parametre effectif : 0)
 6 | Arret := (CPT=nb_stop)
 7 | Csttrue := true
 8 | Csttrue ^= Arret
 9 | ARRET_MANUEL := Csttrue when Arret

*)

Section gestion_echanges.

(*=============================================================================================*)
(*                            Definitions of predicates                                        *)
(*=============================================================================================*)

Variable nb_stop : nat.

Definition cond_arret := [C:(Signal nat)][A:Clock]
((hd C) = (present nb_stop)) -> ((hd A) = (present tt)).

Definition cond_continue := [C:(Signal nat)][A:Clock]
((hd A) = (present tt)) -> ((hd C) = (present nb_stop)).

(* Counter of the consecutive synchronous moments between two signals *)

CoFixpoint cpt_inst_synch :
(U,V:Set)nat->(Signal U)->(Signal V)->(Signal nat) :=
[U,V:Set][n:nat][X:(Signal U)][Y:(Signal V)]Cases X Y of
 (Cons absent X')      (Cons absent Y')      => (Cons (absent nat) (cpt_inst_synch n X' Y'))
|(Cons (present _) X') (Cons absent Y')      => (Cons (present O) (cpt_inst_synch O X' Y'))
|(Cons absent X')      (Cons (present _) Y') => (Cons (present O) (cpt_inst_synch O X' Y'))
|(Cons (present _) X') (Cons (present _) Y') => (Cons (present (S n))
                                                      (cpt_inst_synch (S n) X' Y'))
end.

(*=============================================================================================*)
(*                       Definitions of hypotheses and variables                               *)
(*=============================================================================================*)

Variables CPT,ZCPT,Cst0 : (Signal nat).
Variables H,STOP,ARRET_MANUEL,Csttrue : Clock.
Variable Arret : (Signal bool).

Hypothesis Hnb_stop  : (gt nb_stop O).
Hypothesis H_H       : (OnlyFiniteAbsent H).
Hypothesis Equation0 : (OrderClock STOP H).
Hypothesis Equation1 : (Synchro CPT H).
Hypothesis Equation2 : (Constant O Cst0).
Hypothesis Equation3 : (Synchro Cst0 H).
Hypothesis Equation4 : CPT = (SignalAA_to_SignalA
```

```
                              (default (when (fonction1 [n:nat](plus n (S O)) ZCPT)
                                             (Clock_to_Signal_bool STOP))
                                       (when Cst0
                                             (Clock_to_Signal_bool H))) ).
Hypothesis Equation6 : Arret = (fonction1 [n:nat](beq_nat n nb_stop) CPT).
Hypothesis Equation7 : (Constant tt Csttrue).
Hypothesis Equation8 : (Synchro Csttrue Arret).
Hypothesis Equation9 : ARRET_MANUEL = (when Csttrue Arret).

(*=============================================================================================*)
(*          Hypotheses for preliminary lemmas : Parameter of initialization                    *)
(*=============================================================================================*)

Section Lemmes_gestion_echanges.

Variable ni : nat.
Hypothesis Equation5 : ZCPT = (pre ni CPT).

(*---------------------------------------------------------------------------------------------*)
(*    Lemma 1 : a stop order is sent as soon as the counter's value is nb_stop                 *)
(*---------------------------------------------------------------------------------------------*)

Lemma gest_errs1 : (Globally2 cond_arret CPT ARRET_MANUEL).

(*---------------------------------------------------------------------------------------------*)
(*  Lemma 1b : when a stop order is sent, the counter's value is nb_stop                       *)
(*---------------------------------------------------------------------------------------------*)

Lemma eq_beq_nat : (n,m:nat)n=m->(beq_nat n m)=true.

Lemma beq_nat_eq : (n,m:nat)(beq_nat n m)=true->n=m.

Lemma gest_errs1b : (Globally2 cond_continue CPT ARRET_MANUEL).

(*---------------------------------------------------------------------------------------------*)
(* Lemma 2 : CPT is a counter of the consecutive synchronous moments between H and STOP        *)
(*---------------------------------------------------------------------------------------------*)

Lemma gest_errs2 : CPT = (cpt_inst_synch ni H STOP).

(*---------------------------------------------------------------------------------------------*)
(* Lemma 3 : We have a counter of the consecutive synchronous moments between H and STOP,      *)
(*           and when the value of this counters nb_stop, a stop order is sent.                *)
(*           A stop order is sent only when this value is nbstop.                              *)
(*---------------------------------------------------------------------------------------------*)

Lemma gest_errs3 :
(CPT = (cpt_inst_synch ni H STOP)) /\
(Globally2 cond_arret CPT ARRET_MANUEL) /\ (Globally2 cond_continue CPT ARRET_MANUEL).

End Lemmes_gestion_echanges.

(*=============================================================================================*)
```

```
(*   Theorems : instanciation of the parameters of initialization                    *)
(*====================================================================================*)

Hypothesis Equation5 : ZCPT = (pre 0 CPT).

Theorem gest_errs :
(CPT = (cpt_inst_synch 0 H STOP)) /\
(Globally2 cond_arret CPT ARRET_MANUEL) /\ (Globally2 cond_continue CPT ARRET_MANUEL).

End gestion_echanges.
```

B.3 Dynamique

The Dynamique library contains proofs of the properties of the DYNAMIQUE process.

```
Require Ext_ZArith.
Require Classical.
Require Temporal.
Require OrderClock.
Require Flot.

Implicit Arguments On.

(*---------------------------------- DYNAMIQUE PROCESS ---------------------------------*)
(*

   | NIVEAU_CAL_MIN ^= NIVEAU_CAL_MAX
   | NIVEAU_CAL_MIN := ((NIVEAU_AJ_MIN-(VAPEUR_AJ_MAX*Delta_t))
                       - (0.5*U1*Delta_t*Delta_t))+(P*Delta_t)
   | NIVEAU_CAL_MAX := (NIVEAU_AJ_MAX-(VAPEUR_AJ_MIN*Delta_t))
                       + (0.5*U2*Delta_t*Delta_t)+(P*Delta_t)
   | NMAX := NIVEAU_CAL_MAX$1 init C
   | NMIN := NIVEAU_CAL_MIN$1 init 0.0

   | VAPEUR_CAL_MIN ^= VAPEUR_CAL_MAX
   | VAPEUR_CAL_MIN := VAPEUR_AJ_MIN-(U2*Delta_t)
   | VAPEUR_CAL_MAX := VAPEUR_AJ_MAX+(U1*Delta_t)
   | VMAX := VAPEUR_CAL_MAX$1 init 0.0
   | VMIN := VAPEUR_CAL_MIN$1 init 0.0

   | NIVEAU_AJ_MIN := (NIVEAU when JAUGE_OK) default NMIN
   | NIVEAU_AJ_MAX := (NIVEAU when JAUGE_OK) default NMAX

   | VAPEUR_AJ_MIN := (VAPEUR when UMDV_OK) default VMIN
   | VAPEUR_AJ_MAX := (VAPEUR when UMDV_OK) default VMAX

Equations :

P00| VAPEUR ^= NIVEAU
P01| JAUGE_OK ^= NIVEAU
P02| UMDV_OK ^= NIVEAU
P03| P ^= NIVEAU
P04| NIVEAU_AJ_MIN ^= NIVEAU
```

```
P05| VAPEUR_AJ_MIN ^= NIVEAU
P06| NIVEAU_AJ_MAX ^= NIVEAU
P07| VAPEUR_AJ_MAX ^= NIVEAU
P08| NIVEAU_CAL_MIN ^= NIVEAU
P09| VAPEUR_CAL_MIN ^= NIVEAU
P10| NIVEAU_CAL_MAX ^= NIVEAU
P11| VAPEUR_CAL_MAX ^= NIVEAU
P12| AUX1 ^= NIVEAU
P13| AUX2 ^= NIVEAU
P14| AUX3 ^= NIVEAU
P15| AUX4 ^= NIVEAU

20 | AUX1 := NIVEAU_AJ_MIN-(VAPEUR_AJ_MAX*Delta_t)
21 | AUX2 := AUX1+(P*Delta_t)
22 | 2*NIVEAU_CAL_MIN := 2*AUX2-(U1*Delta_t*Delta_t)
23 | AUX3 := NIVEAU_AJ_MAX-(VAPEUR_AJ_MIN*Delta_t)
24 | AUX4 := AUX3+(P*Delta_t)
25 | 2*NIVEAU_CAL_MAX := 2*AUX4+(U2*Delta_t*Delta_t)
28 | NMAX := NIVEAU_CAL_MAX$1 init NMi     (parametre effectif : C)
29 | NMIN := NIVEAU_CAL_MIN$1 init Nmi     (parametre effectif : 0)

30 | VAPEUR_CAL_MIN := VAPEUR_AJ_MIN-(U2*Delta_t)
31 | VAPEUR_CAL_MAX := VAPEUR_AJ_MAX+(U1*Delta_t)
34 | VMAX := VAPEUR_CAL_MAX$1 init VMi     (parametre effectif : 0)
35 | VMIN := VAPEUR_CAL_MIN$1 init Vmi     (parametre effectif : 0)

40 | NIVEAU_AJ_MIN := (NIVEAU when JAUGE_OK) default NMIN
41 | NIVEAU_AJ_MAX := (NIVEAU when JAUGE_OK) default NMAX

50 | VAPEUR_AJ_MIN := (VAPEUR when UMDV_OK) default VMIN
51 | VAPEUR_AJ_MAX := (VAPEUR when UMDV_OK) default VMAX

*)

Section Dynamique.

(*=============================================================================================*)
(*                 Definitions of the predicates applied to Globally                           *)
(*=============================================================================================*)

Definition inf_strict := [X:(Signal Z)][Y:(Signal Z)]
(x,y:Z)(hd X)=(present x)->(hd Y)=(present y)->(Zlt x y).

Definition inf_egal := [X:(Signal Z)][Y:(Signal Z)]
(x,y:Z)(hd X)=(present x)->(hd Y)=(present y)->(Zle x y).

(*=============================================================================================*)
(*                   Definitions of hypotheses and variables                                   *)
(*=============================================================================================*)

Variables C,W,U1,U2,Delta_t,deux : Z.
Variables VAPEUR,NIVEAU,P,AUX1,AUX2,AUX3,AUX4 : (Signal Z).
Variables NIVEAU_CAL_MIN,NIVEAU_CAL_MAX,NMIN,NMAX : (Signal Z).
```

```
Variables VAPEUR_CAL_MIN,VAPEUR_CAL_MAX,VMIN,VMAX : (Signal Z).
Variables NIVEAU_AJ_MIN,NIVEAU_AJ_MAX,VAPEUR_AJ_MIN,VAPEUR_AJ_MAX : (Signal Z).
Variables UMDV_OK,JAUGE_OK : (Signal bool).

Hypothesis Hdeux : deux = '2'. (* to disable implicit simplifications *)
Hypothesis HC  : (Zlt '0' C).
Hypothesis HW  : (Zlt '0' W).
Hypothesis HU1 : (Zlt '0' U1).
Hypothesis HU2 : (Zlt '0' U2).
Hypothesis HDt : (Zlt '0' Delta_t).

Hypothesis P00 : (Synchro VAPEUR NIVEAU).
Hypothesis P01 : (Synchro JAUGE_OK NIVEAU).
Hypothesis P02 : (Synchro UMDV_OK NIVEAU).
Hypothesis P03 : (Synchro P NIVEAU).
Hypothesis P04 : (Synchro NIVEAU_AJ_MIN NIVEAU).
Hypothesis P05 : (Synchro VAPEUR_AJ_MIN NIVEAU).
Hypothesis P06 : (Synchro NIVEAU_AJ_MAX NIVEAU).
Hypothesis P07 : (Synchro VAPEUR_AJ_MAX NIVEAU).
Hypothesis P08 : (Synchro NIVEAU_CAL_MIN NIVEAU).
Hypothesis P09 : (Synchro VAPEUR_CAL_MIN NIVEAU).
Hypothesis P10 : (Synchro NIVEAU_CAL_MAX NIVEAU).
Hypothesis P11 : (Synchro VAPEUR_CAL_MAX NIVEAU).
Hypothesis P12 : (Synchro AUX1 NIVEAU).
Hypothesis P13 : (Synchro AUX2 NIVEAU).
Hypothesis P14 : (Synchro AUX3 NIVEAU).
Hypothesis P15 : (Synchro AUX4 NIVEAU).

Hypothesis Eq20 : (Op Zminus
                     NIVEAU_AJ_MIN
                     (fonction1 [n:Z](Zmult Delta_t n) VAPEUR_AJ_MAX)
         AUX1).
Hypothesis Eq21 : (Op Zplus
      AUX1
      (fonction1 [n:Z](Zmult Delta_t n) P)
      AUX2).
Hypothesis Eq22 : (fonction1 [n:Z](Zmult deux n) NIVEAU_CAL_MIN) =
                  (fonction1 [n:Z](Zminus n (Zmult U1 (Zmult Delta_t Delta_t)))
                             (fonction1 [n:Z](Zmult deux n) AUX2)).
Hypothesis Eq23 : (Op Zminus
                     NIVEAU_AJ_MAX
                     (fonction1 [n:Z](Zmult Delta_t n) VAPEUR_AJ_MIN)
      AUX3).
Hypothesis Eq24 : (Op Zplus
      AUX3
      (fonction1 [n:Z](Zmult Delta_t n) P)
                     AUX4).
Hypothesis Eq25 : (fonction1 [n:Z](Zmult deux n) NIVEAU_CAL_MAX) =
                  (fonction1 [n:Z](Zplus n (Zmult U2 (Zmult Delta_t Delta_t)))
                             (fonction1 [n:Z](Zmult deux n) AUX4)).

Hypothesis Eq30 : VAPEUR_CAL_MIN=(fonction1 [n:Z](Zminus n (Zmult U2 Delta_t)) VAPEUR_AJ_MIN).
Hypothesis Eq31 : VAPEUR_CAL_MAX=(fonction1 [n:Z](Zplus n (Zmult U1 Delta_t)) VAPEUR_AJ_MAX).
```

```
Hypothesis Eq40 : NIVEAU_AJ_MIN=(SignalAA_to_SignalA (default (when NIVEAU JAUGE_OK) NMIN)).
Hypothesis Eq41 : NIVEAU_AJ_MAX=(SignalAA_to_SignalA (default (when NIVEAU JAUGE_OK) NMAX)).

Hypothesis Eq50 : VAPEUR_AJ_MIN=(SignalAA_to_SignalA (default (when VAPEUR UMDV_OK) VMIN)).
Hypothesis Eq51 : VAPEUR_AJ_MAX=(SignalAA_to_SignalA (default (when VAPEUR UMDV_OK) VMAX)).

(*=============================================================================================*)
(*                            Lemmas for solutions of inequations                              *)
(*=============================================================================================*)

Section Lemmes_Inequations.

Lemma Ineq_Vapeur : (a,b:Z) `a <= b` -> `a-U2*Delta_t < b+U1*Delta_t`.

Lemma Ineq_Niveau : (a,b,c,d,e:Z) `a <= b` -> `c <= d` ->
`0 <    deux*(b-Delta_t*c+Delta_t*e)+U2*(Delta_t*Delta_t) +
     (-(deux*(a-Delta_t*d+Delta_t*e)-U1*(Delta_t*Delta_t)))`.

End Lemmes_Inequations.

(*=============================================================================================*)
(*     Hypotheses for preliminary lemmas : Parameter of initialization                         *)
(*=============================================================================================*)

Section Lemmes_Dynamique.

Variables VMi,Vmi,NMi,Nmi: Z.

Hypothesis HVmVM  : (Zle Vmi VMi).
Hypothesis HNmNM  : (Zle Nmi NMi).

Hypothesis Eq28 : NMAX = (pre NMi NIVEAU_CAL_MAX).
Hypothesis Eq29 : NMIN = (pre Nmi NIVEAU_CAL_MIN).

Hypothesis Eq34 : VMAX = (pre VMi VAPEUR_CAL_MAX).
Hypothesis Eq35 : VMIN = (pre Vmi VAPEUR_CAL_MIN).

(*---------------------------------------------------------------------------------------------*)
(*                Relation between VAPEUR_CAL_MIN and VAPEUR_CAL_MAX                           *)
(*---------------------------------------------------------------------------------------------*)

Lemma Vc1_inf_Vc2 : (Globally2 inf_strict VAPEUR_CAL_MIN VAPEUR_CAL_MAX).

(*---------------------------------------------------------------------------------------------*)
(*                Relation between VAPEUR_AJ_MIN and VAPEUR_AJ_MAX                             *)
(*---------------------------------------------------------------------------------------------*)

Lemma Va1_infegal_Va2 : (Globally2 inf_egal VAPEUR_AJ_MIN VAPEUR_AJ_MAX).

(*---------------------------------------------------------------------------------------------*)
(*                Relation between NIVEAU_CAL_MIN and NIVEAU_CAL_MAX                           *)
(*---------------------------------------------------------------------------------------------*)
```

```
Lemma Nc1_inf_Nc2 : (Globally2 inf_strict NIVEAU_CAL_MIN NIVEAU_CAL_MAX).

(*-------------------------------------------------------------------------------------*)
(*               Relation between NIVEAU_AJ_MIN and NIVEAU_AJ_MAX                      *)
(*-------------------------------------------------------------------------------------*)

Lemma Na1_infegal_Na2 : (Globally2 inf_egal NIVEAU_AJ_MIN NIVEAU_AJ_MAX).

End Lemmes_Dynamique.

(*=====================================================================================*)
(*          Theorems : instanciation of the parameters of initialization               *)
(*=====================================================================================*)

Hypothesis Eq28 : NMAX = (pre  C  NIVEAU_CAL_MAX).
Hypothesis Eq29 : NMIN = (pre '0' NIVEAU_CAL_MIN).

Hypothesis Eq34 : VMAX = (pre '0' VAPEUR_CAL_MAX).
Hypothesis Eq35 : VMIN = (pre '0' VAPEUR_CAL_MIN).

(*-------------------------------------------------------------------------------------*)
(*              Relation between VAPEUR_CAL_MIN and VAPEUR_CAL_MAX                     *)
(*-------------------------------------------------------------------------------------*)

Theorem Vcmin_inf_Vcmax : (Globally2 inf_strict VAPEUR_CAL_MIN VAPEUR_CAL_MAX).

(*-------------------------------------------------------------------------------------*)
(*              Relation between VAPEUR_AJ_MIN and VAPEUR_AJ_MAX                       *)
(*-------------------------------------------------------------------------------------*)

Theorem Vamin_infegal_Vamax : (Globally2 inf_egal VAPEUR_AJ_MIN VAPEUR_AJ_MAX).

(*-------------------------------------------------------------------------------------*)
(*              Relation between NIVEAU_CAL_MIN and NIVEAU_CAL_MAX                     *)
(*-------------------------------------------------------------------------------------*)

Theorem Ncmin_inf_Ncmax : (Globally2 inf_strict NIVEAU_CAL_MIN NIVEAU_CAL_MAX).

(*-------------------------------------------------------------------------------------*)
(*              Relation between NIVEAU_AJ_MIN and NIVEAU_AJ_MAX                       *)
(*-------------------------------------------------------------------------------------*)

Theorem Namin_infegal_Namax : (Globally2 inf_egal NIVEAU_AJ_MIN NIVEAU_AJ_MAX).

End Dynamique.
```

Contents

References

[1] Preliminary report of the dagstuhl-seminar 9523.
`http://www.informatik.uni-kiel.de/~procos/dag9523/dagstuhl-report.ps.Z`.

[2] Solutions to the steam boiler case study.
`http://www.informatik.uni-kiel.de/~procos/dag9523/sb-solutions.html`.

[3] Abrial (J.R.), Börger (E.) et Langmaack. (H.). – Formal methods for industrial applications: Specifying and programming the steam boiler control. *Lecture Notes in Computer Science*, vol. 1165, october 1996.

[4] Baker (T. P.) et Pazy (O.). – Real-time features for Ada 9x. *In : Proc. of the IEEE Real-time Systems Symposium*, pp. 172–180.

[5] Barras (B.). – *The COQ Proof Assistant Reference Manual.* – Technical report, INRIA, september 1998.

[6] Benveniste (A.) et Le Guernic (P.). – Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, vol. 16, n2, 1991, pp. 103–149.

[7] Berry (G.) et Gonthier (G.). – The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, vol. 19, n2, 1992, pp. 87–152.

[8] Bournai (P.) et Le Guernic (P.). – *Un environnement graphique pour le langage `Signal`.* – Research Report n741, IRISA, july 1993.

[9] Boussinot (F.) et de Simone (R.). – The esterel language. *Proc. of the IEEE*, vol. 79, n 9, sep 1991, pp. 1293–1304.

[10] Cattel (T.) et Duval (G.). – The steam boiler problem in lustre. *Lecture Notes in Computer Science*, vol. 1165, october 1996.

[11] Giménez (E.). – *Un calcul de constructions infinies et son application à la Vérification des Systèmes Communicants.* – Ph. d. thesis, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, december 1996.

[12] Halbwachs (N.), Caspi (P.), Raymond (P.) et Pilaud (D.). – The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, vol. 79, n9, september 1991, pp. 1305–1320.

[13] Harel (D.). – Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, vol. 8, n3, June 1987, pp. 231–274.

[14] Hoare (C. A. R.). – *Communicating Sequential Processes.* – Prentice-Hall International, 1985.

[15] Le Guernic (P.), Gautier (T.), Le Borgne (M.) et Le Maire (C.). – Programming real-time applications with signal. *Proc. of the IEEE*, vol. 79, n 9, september 1991, pp. 1321–1335.

[16] Milner (R.). – Calculi for synchrony and asynchrony. *Theoretical computer science*, vol. 25, n3, 1983, pp. 267–310.

[17] Nowak (D.). – *Spécification et preuve de systèmes réactifs.* – Ph. d. thesis, IFSIC, Université de Rennes1, 1999.

[18] Smarandache (L.). – *Transformations affines d'horloges: application au codesign de systèmes temps-réels en utilisant les langages Signal et Alpha.* – Ph. d. thesis, IFSIC, Université de Rennes1, 1998.

[19] Werner (B.). – *Une théorie des constructions inductives.* – Ph. d. thesis, Université de Paris VII, may 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399